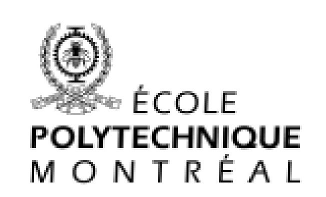




ÉCOLE POLYTECHNIQUE

DE MONTRÉAL

DÉPARTEMENT DE GÉNIE INFORMATIQUE

Interface de contrôle pour joueurs de soccer robotisés

Rapport de projet de fin d'études soumis
comme condition partielle à l'obtention du
diplôme de baccalauréat en ingénierie.

Présenté par : Alexandre Venne
Matricule : 1123187
Directeur de projet : Richard Gourdeau

Date : 16 avril 2004

Sommaire

Développé dans le cadre d'un projet de fin d'étude, ce projet consiste à concevoir une solution logiciel permettant de contrôler plus facilement le comportement des robots joueurs de soccer conçu par le groupe Robofoot de l'École Polytechnique. Auparavant, les gens responsables du développement des robots ne possédaient pas une application graphique simple pour envoyer des commandes au robot, visualiser l'état des robots ou pour changer le comportement (le cerveau) d'un robot. Tout devait se faire en mode console sur une seule ligne de commande. La réalisation de ce projet facilitera donc leur travail.

Le projet comporte deux éléments interdépendants. D'abord, il s'agit de développer une interface usager graphique permettant de manipuler les connexions réseaux vers les robots. Ce logiciel ne fait pas que gérer les connexions, il agit principalement comme un intermédiaire de contrôle entre l'utilisateur et le robot dans ce sens qu'elle peut afficher de façon périodique la valeur des variables critiques du robot, envoyer des commandes au robot et changer facilement son comportement.

Parallèlement, le deuxième élément développé est un modèle d'application serveur intégré à l'application principale de chacun des robots. Cette application permettra évidemment à l'interface client de se connecter à chacun des robots à l'aide d'un lien sans fil en utilisant le protocole Ethernet TCP/IP. L'application serveur contient plusieurs fonctions de service permettant d'exécuter des tâches spécifiques sur le système. Ces fonctions de service sont appelées par un processus concurrent responsable de la réception de message provenant de l'application client.

Le développement de l'interface usager s'est fait en suivant un processus itératif de développement logiciel en utilisant UML où cela était nécessaire. L'interface usager graphique développé est portable sur Win32 et Linux. La librairie graphique wxWidgets a été utilisée pour développer l'interface, tant au niveau des contrôles accessibles à l'utilisateur qu'au niveau des éléments plus abstraits tels que les processus concurrents.

Table des matières

Liste des figures

Remerciements

Je tiens à remercier mon directeur de projet M. Richard Gourdeau, professeur au département de génie électrique, pour son appuie au projet et ainsi que pour la qualité de son enseignement reçu tout au long de mon baccalauréat.

De plus, il m’est impossible de passer sous silence l’appuie généreux de la part de Sylvain Marleau, étudiant à la maîtrise, qui m’a supporté tout au long de ma démarche et à toujours su répondre à mes questions sans la moindre hésitation.

Glossaire

API : *Application programming interface*, ou interface de programmation d'application, est une description des différentes fonctions et entités qu'une bibliothèque logiciel possède, permettant à un programme d'y avoir accès pour faciliter son développement sur une plateforme donnée.

Cerveau : Le cerveau est considéré dans ce travail comme l'objet logiciel permettant de contrôler la logique des robots. Par exemple, il est possible d'avoir un cerveau joueur ou un cerveau gardien sur le robot. Lorsqu'un cerveau est instancié dans l'application du robot, il donne un comportement précis au robot.

Librairie GUI : *Graphical User Interface Librairy*, ou librairie d'interface graphique usager est une bibliothèque logiciel contenant une structure de classe utilisable pour les programmeurs afin de concevoir des interfaces graphiques sur une plateforme donnée. Une interface graphique est considérée comme une image interactive permettant de contrôler une application fonctionnant sur un système d'exploitation à fenêtrage.

MDI : Fonction qui permet à un logiciel d'application d'afficher plus d'une fenêtre de documents simultanément, l'utilisateur pouvant ainsi travailler avec plusieurs documents à la fois ou avoir plusieurs vues d'un même document. Cette fonctionnalité est particulière au système d'exploitation Windows.

Plate-forme : Structure matérielle d'un système informatique, principalement basée sur le type de système d'exploitation utilisé.

Socket : Terme anglophone utilisé pour définir une interface de connexion. Une interface de connexion permet à des applications de communiquer d'un ordinateur à l'autre, indépendamment du type de réseau utilisé. Les interfaces de connexion

fonctionnent toujours par paire; une interface est utilisée pour le poste client et l'autre, pour le serveur.

SSH : Programme permettant d'offrir une communication encryptée sur Internet. Son utilisation permet entre autre de communiquer sous la forme de fenêtre console et de lancer des programmes sur un ordinateur à distance.

Win32 : Win32 est la définition d'une cible architecturale pour les logiciels fonctionnant sur le système d'exploitation Microsoft® Windows.

1 Introduction

Depuis les 30 dernières années, le développement en robotique connaît une croissance très importante. À travers le monde, des chercheurs tentent par tous les moyens de développer des automates de plus en plus intelligents et autonomes, capables d'agir au même niveau que l'homme. Présentement, nous savons que l'automatisation prend déjà beaucoup de place dans notre société, mais les rôles confiés à ces machines sont principalement reliés à l'exécution de tâches répétitives dans un environnement connu. Maintenant, les défis qui attendent les scientifiques sont de mettre ensemble toutes les connaissances reliées à la robotique telles que la vision, l'intelligence artificielle, la communication et le contrôle afin de créer un objet capable de rivaliser la complexité de l'homme. Déjà, la machine a su concurrencer les meilleurs joueurs d'échec au monde. Mais qu'en est-il dans le domaine du sport où le temps de réaction, la précision des mouvements et la collaboration entre les individus sont essentielles? De là, la naissance d'une coupe du monde de joueurs de soccer robotisés, appelé *RoboCup* [5], dont l'objectif est claire : dès l'an 2050, une équipe de joueurs de soccer robotisés autonomes sera capable de vaincre l'équipe humaine championne du monde de soccer.

Le groupe Robofoot [6] de l'École Polytechnique de Montréal travail sur ce projet dans la catégorie des robots rouleurs. La partie mécanique des robots est déjà développée et chacun de ceux-ci est constitué d'un ordinateur embarqué avec le système d'exploitation Linux. Le lien de communication entre les différents robots est établi grâce à une carte de communication sans fil sur chacun de ceux-ci. Auparavant, les gens responsables au développement devaient lancer manuellement le logiciel de contrôle des robots (le cerveau des robots) en établissant une communication avec SSH et en définissant les paramètres du robot directement sur la ligne de commande. Évidemment, cette tâche devenait rapidement ardu lorsque venait le temps de démarrer les six robots ou simplement lorsqu'un changement de cerveau était nécessaire. L'objectif de ce projet est dont de développer une solution logiciel basée sur une architecture client-serveur qui permettra de contrôler plus facilement le comportement et les paramètres donnés aux robots. Ce projet est donc décomposé en deux éléments dépendants. Il s'agit de concevoir un modèle d'application

serveur, où chacun des robots en est l'hôte. Développée sur une base déjà existante, cette application permettrait de répondre au besoin du groupe en ce qui à trait à l'interaction avec les robots. Le projet est aussi constitué du développement d'une application client, ou plutôt une interface de contrôle permettant de communiquer avec l'application serveur de chacun des robots. Toutefois, il est important de saisir que l'application client doit être une interface usager graphique fonctionnant aussi bien sur une plateforme Win32 que Linux. C'est en partie à ce niveau que le projet prend tout son sens.

Ce rapport est donc un résumé de la démarche suivie pour la réalisation de ce projet. Le premier élément présenté est la problématique. Dans cette section, le projet est expliqué dans son contexte en faisant le point sur la situation actuelle. Ceci permet de bien comprendre l'ampleur de la problématique avec les principales difficultés à surmonter, tant au niveau de l'application client qu'au niveau de l'application serveur. De plus, les deux objectifs du projet y sont présentés de façon claire. Cette section nous amène aussi à comprendre tout l'intérêt que le groupe Robofoot porte à la réalisation de ce travail.

Dans un deuxième temps, la méthodologie utilisée pour atteindre les objectifs du projet est exposée. L'élaboration de la méthodologie nous amène à comprendre l'utilité d'appliquer une méthode de conception de logiciel dans un tel contexte. Chacune des étapes du processus de conception telles la prise des besoins, l'analyse et conception, la programmation et les tests, sont décrites et forment le chemin à suivre pour la construction de l'application. Les diverses justifications à propos du choix des outils et des méthodes susceptibles de faire progresser le projet vers une solution faisable, notamment au niveau du choix de la librairie graphique utilisée et des moyens de communication privilégiés, s'ajoutent enfin au contenu du chapitre.

Les résultats obtenus suite à l'application de la méthode de conception forme le contenu du troisième chapitre. Cette section expose en détail toutes les parties du logiciel client et du logiciel serveur. D'abord, ces résultats sont énoncés à partir des diagrammes et des analyses définies lors de l'étape de conception. L'explication des résultats se concentre principalement sur l'architecture du logiciel proposé à l'aide de diagrammes de classes ou

de séquences. Ensuite, des captures d’écran représentant l’interface de l’application client ainsi qu’un parallèle avec les fonctionnalités du logiciel y est présentés.

Enfin, un cours chapitre de discussion contiendra un retour en arrière sur la démarche utilisé et sur l’ensemble des travaux réalisés à la lumière de mes attentes initiales. La méthode de développement utilisée et le choix des outils font face à une critique afin que l’expérience de la réalisation de ce projet soit encore plus grande.

2 Problématique

Dans ce chapitre, le projet est expliqué dans son contexte en faisant le point sur la situation actuelle. À la lumière de cette situation, une problématique est dégagée sous la forme de cas d'utilisation.

2.1 Situation actuelle

Tel que mentionné auparavant, le groupe Robofoot cherche à développer des robots autonomes capables de jouer au soccer. À l'heure actuelle, le groupe de travail est capable de faire fonctionner six robots de manière autonome. Sans vouloir entrer complètement dans les détails de la structure des robots, disons simplement que chacun des robots est un ordinateur en soit. Les éléments qui nous intéressent dans la définition de la problématique sont que chacun des robots a un disque rigide pour stocker des programmes et une carte de réseau 802.11b pour communiquer entre eux ou avec un ordinateur faisant partie du même réseau. La partie logicielle des robots est celle qui s'occupe du contrôle et de la prise de décision. Tous les éléments logiciels implantés sur le robot fonctionnent grâce au système d'exploitation Linux. La figure suivante montre les différents concepts lors la situation initiale.

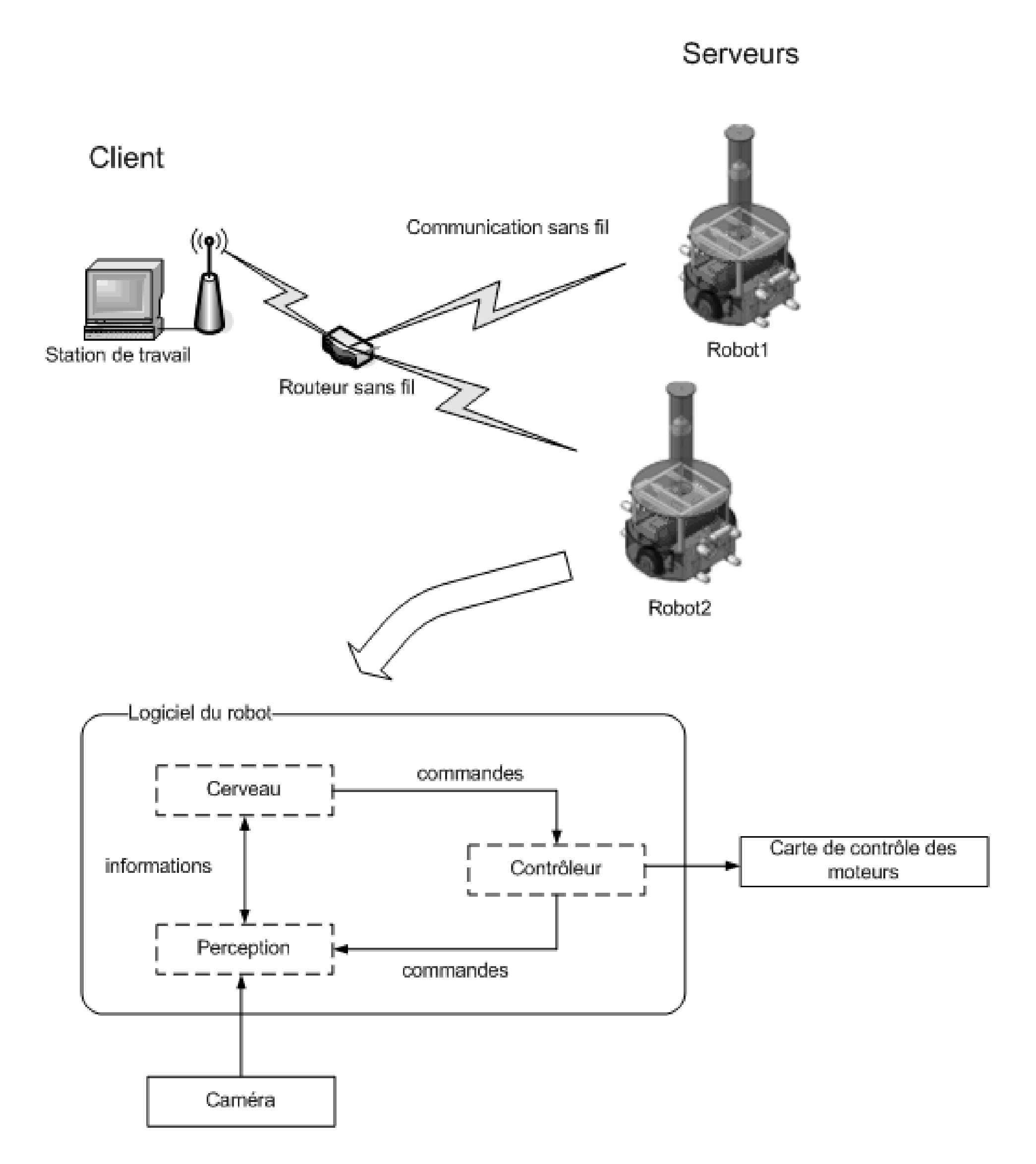


Figure 2.1 - Concepts de la situation initiale

Comme il est possible de le constater à la figure 2.1, l’application de contrôle des robots est comparable à un vase presque clos. Globalement, ce logiciel est séparé en trois concepts: le module de cerveau, le module perception et un module contrôleur. Entre chacun de ces modules, il y a échange d’informations ou de commandes. Le fait que le logiciel soit défini comme un contenant hermétique est à la base de notre problématique. Au démarrage de l’ordinateur du robot, le système d’exploitation initialise toutes les périphériques nécessaires à son fonctionnement pour ensuite tomber en mode d’attente d’une connexion. Dans l’architecture actuelle, la station de travail agit comme un client et

les robots sont des serveurs puisqu'ils doivent répondre à des requêtes du client. Pour faire fonctionner le robot, il suffit évidemment de démarrer son programme. Les utilisateurs doivent donc ouvrir une session *SSH* sur la station de travail afin d'établir une connexion en mode console au système d'exploitation du robot. Ensuite, il faut faire démarrer le programme pour que le robot s'active. Cette tâche est évidemment fastidieuse puisqu'il faut faire cette démarche pour les six robots à activer. Aussi, mentionnons que les paramètres donnés sur la ligne de commande pour le démarrage de l'application définissent le comportement futur du robot. Ainsi, puisque le cerveau d'un robot doit être différent selon si sa logique est celle d'un joueur attaquant ou d'un gardien de but, il est impossible de modifier son comportement sans avoir préalablement redémarrer l'application. Le fait d'être incapable de modifier dynamiquement le cerveau du robot représente une première limite du système actuel et forme la partie initiale de la problématique.

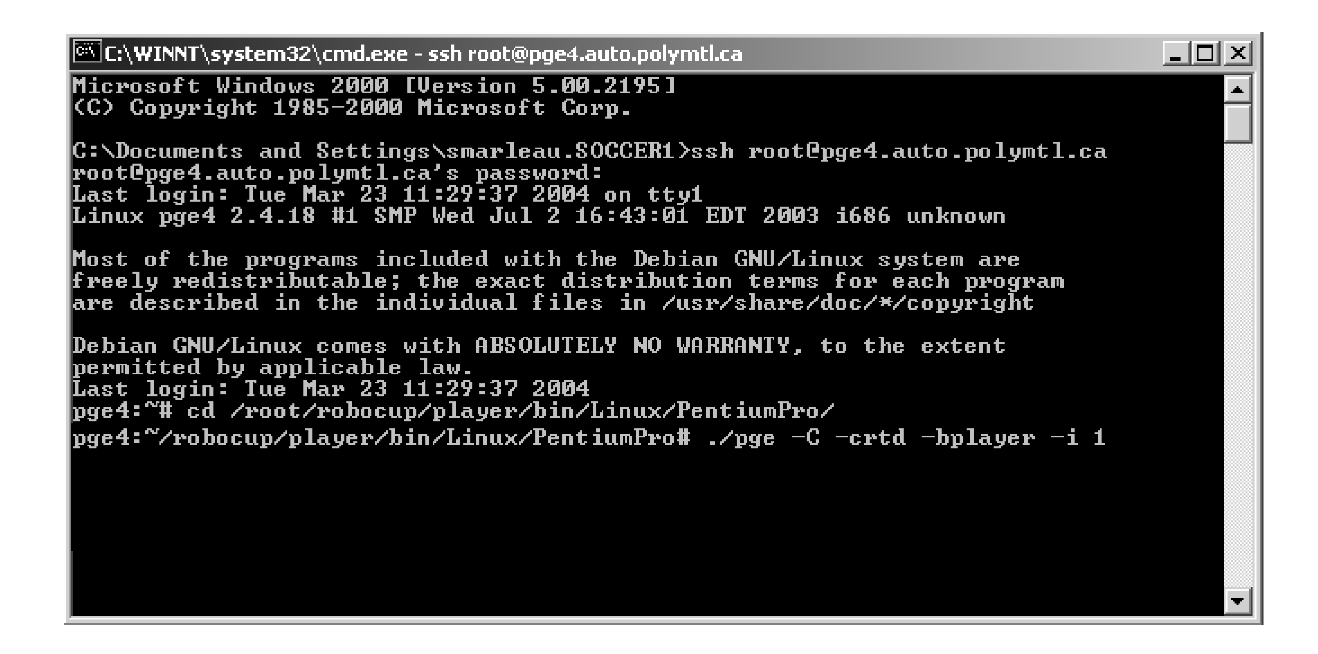


Figure 2.2 - Démarrage du programme d'un robot avec tous les paramètres sur la ligne de commande

Une deuxième limite de la situation actuelle concerne la visualisation des différentes valeurs des variables du programme du robot. Pour être capable de connaître l'état d'un robot dans le déroulement de sa logique ou pour simplement obtenir la valeur d'une variable critique du système telle que sa position, il faut aller lire une variable directement dans l'espace mémoire du logiciel du robot. Avec l'architecture initiale, sans inscrire

régulièrement la valeur sur la console, c'est-à-dire en utilisant la fonction *printf*, il est impossible d'obtenir cette information. Cette pratique réduit considérablement la bande passante et limite sur le nombre de variables pour lesquelles les développeurs veulent effectuer un suivi lors du déverminage de l'application. Pour bien comprendre l'effet de cette problématique, il suffit de s'imaginer de quoi aurait l'air la fenêtre de console de la figure 2.2, avec l'écriture de la valeur de plusieurs variables à l'écran.

Une autre limite concerne l'envoie de commandes dynamiques au robot à partir de la station de travail. Lorsqu'un robot est en mode exécution, il est impossible présentement de lui faire exécuter une commande particulière sans passer par un serveur de manette de jeux. Donc, la liste des commandes possibles à exécuter est limitée par l'utilisation d'une manette. Bref, il est impossible de définir une liste dynamique de commandes qui pourraient être lancées à partir de la console ou d'une toute autre interface. Notons rapidement qu'un exemple de commande à donner à un des robots pourrait être l'une des suivantes : *Start, Stop, Kick, GotoXY* ou *Spin.* Évidemment, le nombre de commandes possibles devra être infini, il suffit simplement que le cerveau soit capable de les interpréter.

Enfin, une dernière limite, quoi que moins contraignante, concerne l'utilisation d'un serveur de manette de jeux optionnel au contrôle des robots. Actuellement, le logiciel implanté sur les robots doit constamment demander l'état de la manette de jeux. En effectuant des requêtes continues, la bande passante nécessaire à la communication est réduite et le délai de propagation de l'information entre la manette et le serveur peut être assez important lorsque le réseau sans fil est encombré. Donc, selon de la rapidité des actions imposées à la manette, certaines commandes peuvent être perdues.

2.2 Problématique à résoudre

Suite à l'énumération de toutes les limites, il est possible de dégager une problématique claire et simple. La problématique du projet est de développer une solution

fiable pour enrayer les limites actuelles du système utilisé par le groupe Robofoot. Nous voyons que ces limites se situent tant au niveau du logiciel client exécuté sur la station de travail qu'au niveau de l'architecture en vase clos du programme des robots. L'objectif majeur de ce travail se situe au niveau de la station de travail. Il faut développer une interface graphique portable Linux et Win32, permettant de communiquer facilement avec les robots. Une interface graphique permettrait d'effectuer les manipulations de connexion plus facilement en plus de visualiser les variables, envoyer des commandes et changer la définition d'un cerveau. Le développement d'une telle application entraîne nécessairement des modifications au niveau de la structure du logiciel des robots. Pour être capable d'obtenir une communication fluide entre l'interface de contrôle et le programme des robots, il faut modifier la structure de l'application des robots en y incorporant un élément serveur répondant aux requêtes de l'interface client.

Le deuxième objectif de ce projet est de présenter un modèle d'application serveur fonctionnant avec l'interface client. Plus précisément, l'application serveur (le serveur d'interface) doit permettre la transmission des commandes reçues de l'interface au cerveau du robot, d'envoyer à l'interface client les valeurs de différentes variables et de changer le cerveau courant à la demande de l'utilisateur. Évidemment, l'application doit s'intégrer facilement à la structure actuelle du logiciel des robots. Mentionnons que la majorité des classes nécessaires à la communication au niveau serveur ont déjà été développées précédemment à ce projet, réduisant considérablement la tâche de développement pour l'application serveur. Puisque le domaine du problème est principalement logiciel, le résumé de la problématique s'exprime aussi de la même façon qu'un besoin en notation UML [13], avec des cas d'utilisation. L'expression de la problématique se retrouve donc dans les figures suivantes.

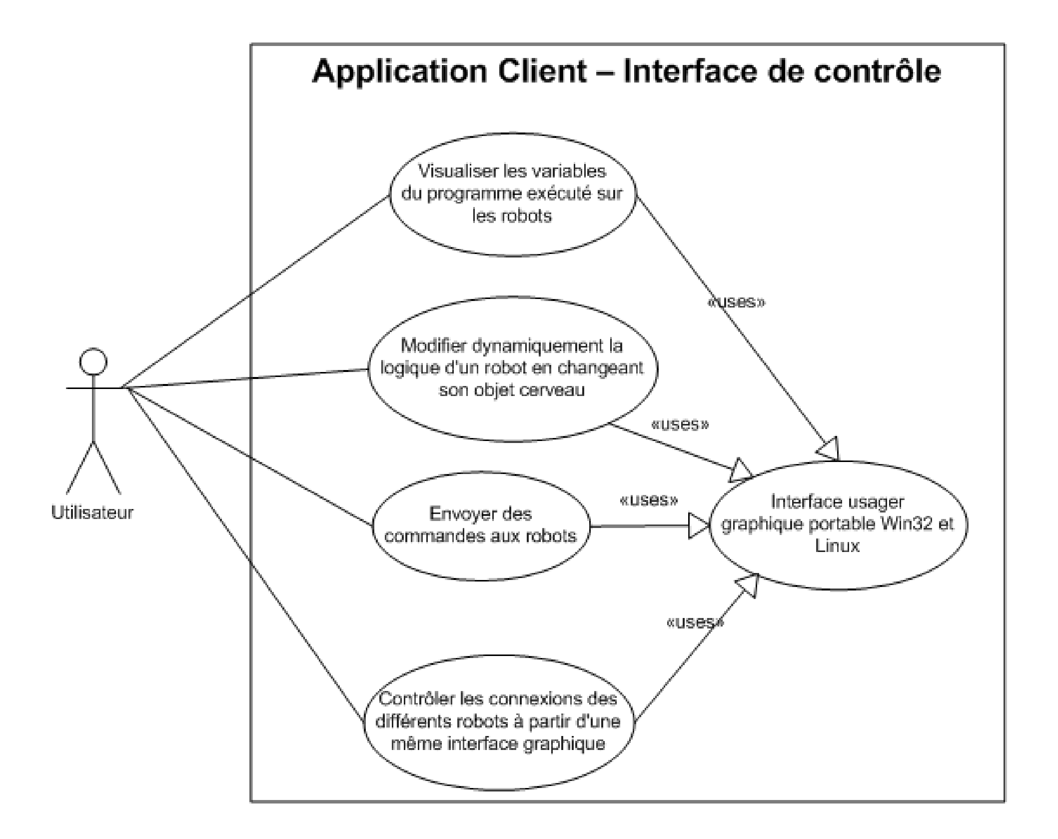


Figure 2.3 - Description sommaires de l'utilisation de l'application client

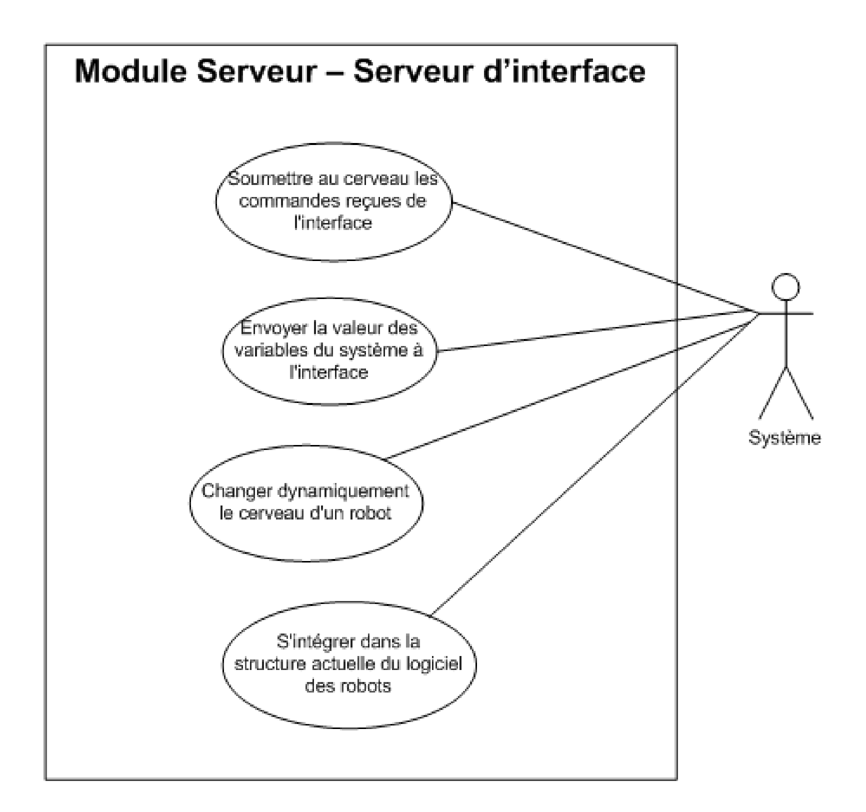


Figure 2.4 - Description sommaire des fonctionnalités de l'application serveur

Les cas d'utilisation précédents expriment globalement ce que le système devra être capable de faire à la fin du projet, tant au niveau de l'interface de contrôle du client qu'au niveau du serveur d'interface implanté dans le programme des robots. Il est important de mentionner qu'une partie du développement de la communication TCP/IP au niveau du serveur a déjà été l'objet d'un développement antérieur. Par conséquent, la problématique se veut davantage un développement de l'interface usager graphique et de la structure du modèle serveur.

L'intérêt dans le développement de ce projet est bénéfique pour le groupe Robofoot, puisqu'il leur permettra de contrôler beaucoup plus facilement leurs robots et ajoutera certainement un outil supplémentaire pour les aider dans leur développement. D'un point de vue personnel, ce projet représente un défi intéressant puisqu'il implique des notions de développement d'interface multi plateforme Win32 et Linux, d'exécution concurrente (*threading*) et de communication par *socket*.

3 Méthodologie

Ce chapitre porte sur la méthodologie adoptée afin de résoudre la problématique. Dans un premier temps, le processus de conception logiciel est expliqué et ensuite, les choix des méthodes et des outils de développement sont justifiés. Enfin, certains points sont clarifiés en ce qui concerne la stratégie de communication.

3.1 Processus de conception logiciel

Un processus de conception logiciel est une méthode de travail proposée en génie logiciel pour développer des applications avec un certain niveau de qualité. Le suivi d'une méthode de conception peut paraître trivial au premier regard, mais il est important de ne pas perdre de vue ses étapes puisque cela peut entraîner des conséquences fâcheuses sur les résultats. Étant donné que le but premier du projet est de concevoir un logiciel, nous considérons important de bien suivre les étapes de développement d'un tel processus, le RAD*(Rapid Application Development)*, afin de s'assurer que le résultat corresponde bien aux besoins initiaux. Les étapes suivies sont dans l'ordre : la prise des besoins, l'analyse et la conception, la programmation et les tests. Malgré cela, il faut considérer qu'étant donné le temps limité pour le développement du projet, ces étapes n'ont pas fait l'objet d'un développement en profondeur et les extrants produits à chacune des étapes sont surtout sommaires. De plus, le processus favorise un développement en spirale où les étapes sont répétées de façon circulaire jusqu'à ce que le projet atteigne un niveau de satisfaction acceptable. Dans les prochaines lignes, les étapes de la prise des besoins, de l'analyse et des tests seront brièvement expliquées en fonction du projet. Les autres étapes, soient l'étape de conception et de la programmation concernent davantage le contenu du prochain chapitre portant sur les résultats.

La première étape à faire est évidemment la prise des besoins. Cette étape est celle où il y a familiarisation avec les activités du groupe Robofoot et pour laquelle la personne responsable explique en détail l'architecture du logiciel actuellement utilisé. Cela permet

de bien comprendre les besoins ainsi que les objectifs du projet. Les besoins fonctionnels ont été clairement définis dans le chapitre précédent, lequel fait mention de la description de la problématique. Cependant, sachant très bien qu'une partie du projet nécessitait le développement d'une interface graphique, il est difficile d'évaluer spécifiquement les besoins quant à la disposition des objets de l'interface. C'est pourquoi un prototype d'interface usager graphique est développé. Un tel prototype permet d'associer certaines fonctionnalités critiques de l'application à une interface usager. Souvent, un prototype permet de faire le lien entre l'étape de la prise des besoins et de l'analyse.

L'étape de l'analyse amène généralement à faire ressortir les différents concepts soulevés lors de la recherche des besoins. En prototypant l'interface usager graphique à l'aide d'un outil de développement rapide tel que Microsoft® Visual Basic© 6.0, nous avons été en mesure de confirmer les besoins des utilisateurs en plus de remarquer que cette pratique ouvre la voie vers de nouvelles discussions. Par exemple, pour la disposition des fenêtres des joueurs dans l'interface, nous nous sommes rendus compte qu'il devenait essentiel d'utiliser un fenêtrage de type MDI. Ce type de fenêtrage amène plusieurs avantages par rapport au fenêtrage standard. D'abord, toutes les fenêtres créées, qui sont des enfants d'une fenêtre parent de type MDI, sont contenues dans la fenêtre parent et l'usager peut les disposer comme bon lui semble. Cette gestion des fenêtres, en plus d'être plus pratique et facile, propose un nombre de connexions illimitées vers les robots à partir d'un seul élément de la barre des tâches du système d'exploitation. Notons au passage que le fenêtrage MDI est une particularité du système d'exploitation Windows et que l'équivalent MDI sous Linux avec GTK+ est une fenêtre avec des onglets.

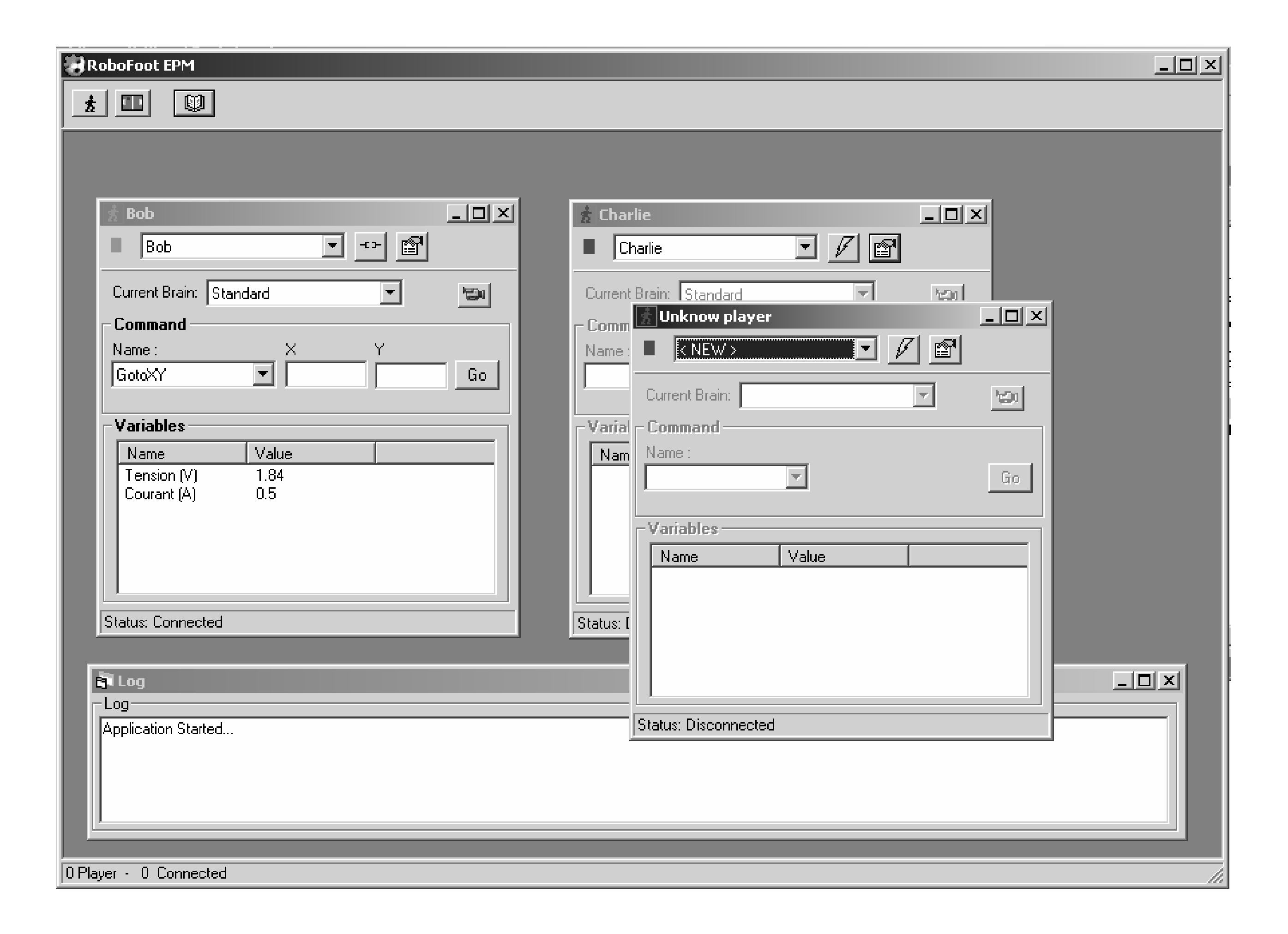


Figure 3.1 - Prototype d'interface avec fenêtre MDI développé en utilisant le logiciel Visual Basic 6.0

Rappelons que le prototype ne fait que proposer un modèle d'interface et ne contient pas d'éléments fonctionnels. Il permet simplement de confirmer certain comportement désiré, mais n'est pas développé dans une optique de performance. Il permet simplement de s'assurer de certaines décisions de l'interface usager qui peuvent avoir une conséquence directe sur la structure des classes de l'application à développer.

Cela nous amène donc à discuter de l'étape de la conception. Cette étape donne naissance principalement à un diagramme de classes, tant pour le logiciel d'interface de la station de travail que pour l'application serveur des robots. Ces diagrammes de classes contiennent la structure du logiciel. Le chapitre sur les résultats explique davantage les structures choisies. Il est important de noter que cette structure est fortement influencée par

les choix des différentes librairies utilisées et par la sélection du langage de programmation approprié. Ainsi, même si aucune ligne de code n'a été inscrite dans le projet, la structure des classes est intimement reliée au choix des librairies utilisées.

Pour la phase de tests, la méthodologie proposée est principalement celle des tests en boîte noire, c'est-à-dire celle où seulement les fonctionnalités implantées sont testées de façon individuelle ainsi que dans l'ensemble. Une révision complète du code aurait certainement amélioré la qualité du produit, mais étant donné le temps limité pour la réalisation de ce projet, le produit n'a pas subit ce genre de tests.

3.2 Choix de l'outil de développement

Lorsque vient le temps de développer une application nécessitant une interface usager graphique, plusieurs options d'outils de développement s'offrent à nous. Si la plate-forme cible est simplement Win32, un langage de développement tel que Visual Basic .NET [8] ou C#.NET aurait simplement fait l'affaire. Toutefois, la problématique se complexifie davantage puisqu'un des critères du projet impose un développement d'interface portable autant sous Win32 que sous Linux. Ceci nous enlève donc le choix d'utiliser la majorité des langages de 4ième génération, forte par leur facilité de développement d'interface usager. Par conséquent, pour optimiser la compatibilité de langage peu importe la plateforme et pour éviter d'utiliser un intermédiaire à l'exécution tel qu'une machine virtuelle Java, le développement en C++ a été favorisé. De plus, ce langage est largement utilisé dans tout le développement fait par les gens de l'équipe et à travers la communauté universitaire. Les ressources capables de maintenir le projet ne sont donc pas limitées. Pour le choix du compilateur, afin de s'éviter bien des maux de tête, notamment avec STL(*Standard Template Library)*, l'utilisation de Visual C++ 7.0 sous Windows est inévitable lors du développement de code portable nécessitant le respect du standard C++. Du côté du système d'exploitation Linux, le compilateur utilisé est g++ de la distribution RedHat 7.3 (noyau 2.4.18-3). GKT+ version 1.2 (minimum) doit être installé avec la

distribution Linux afin de profiter de toutes les possibilités offertes par la librairie graphique utilisée dans ce projet.

3.3 Choix de la librairie graphique (GUI) portable

Maintenant que le choix du langages est justifié, il suffit maintenant de regarder les différentes librairies graphiques disponibles pour développer des interfaces graphiques usager portable en C++. Pour répondre aux besoins, la librairie doit minimalement être capable de fournir des classes pour la programmation réseau à l'aide de *socket*, supporter le fenêtrage de type MDI, fournir une suite d'objets fenêtres habituel pour les interfaces usagers (bouton, liste déroulante, icônes, etc…) et évidemment, supporter le développement impliquant des processus concurrents. Enfin, un autre critère d'importance est que l'utilisation de cette librairie doit être gratuite.

Une première possibilité est d'utiliser la librairie QT [7] de la compagnie Trolltech. Cette librairie est certainement l'une des plus populaires et celle dont le développement est le plus avancé. Plusieurs applications connues ont été développées en utilisant cette librairie multi-plateforme. Pourtant, son utilisation entraîne un inconvénient majeur : elle n'est pas gratuite sous Win32. Par conséquent, cette solution doit être rejetée.

Une autre possibilité est d'utiliser la librairie wxWidgets[9] dont la licence d'utilisation proposé est GPL (*GNU General Public Licences*)[11], qui, par conséquent, donne la possibilité à ses utilisateurs de l'utiliser sans frais. Cette librairie, malgré que son développement soit un peu moins mature que QT, offre plusieurs possibilités intéressantes. Contrairement à QT, wxWidgets est développé par la communauté et chapeauté par l'auteur Julian Smart. Par conséquent, la documentation disponible est comparable à celle qu'offre Trolltech pour sa librairie QT, mais il faut être capable d'utiliser les forums de discussion pour compléter certaines informations, car les subtilités de la libraire ne sont pas toujours évidentes.

Il existe d'autres librairies gratuites pour le développement d'interfaces usagers en C++. Cependant, celles-ci n'offrent pas nécessairement autant de possibilités que QT ou wxWidgets. Ainsi, à la lumière des dernières affirmations, l'utilisation de la librairie wxWidgets, version 2.4.2, est celle qui semble la plus appropriée pour la réalisation du projet.

L'étude des possibilités de cette librairie est nécessaire avant d'être en mesure de concevoir la structure de l'application. Il donc important de mentionner qu'un investissement considérable en temps est nécessaire avant de bien comprendre ses limites et son fonctionnement. Cela peut avoir un impact sérieux sur des décisions structurales de l'application. Cette démarche d'apprentissage a été faite avant la réalisation du projet et les résultats ont été bénéfiques pour l'étape de conception.

3.4 Choix sur la méthode de communication entre les applications

Ce projet nécessite la communication entre une application serveur et une interface usager considérée dans notre cas comme client. Essentiellement, les données sont transférées à l'aide de *sockets* avec TCP/IP sur le réseau local du comité. Toutefois, le contenu du message envoyé entre l'interface et le serveur suggère le choix d'un format de communication. À première vue, le format XML pour communiquer le message aurait pu être intéressant. Sauf que ce format aurait nécessité le traitement des balises tant au niveau du serveur que de l'interface client en plus d'alourdir la taille des message, ayant pour effet d'augmenter la bande passante nécessaire à l'envoie du message.

La stratégie de communication adoptée est celle qui a été développée par Sylvain Marleau. Elle suggère essentiellement l'utilisation de services, enregistrés à la compilation auprès du serveur. Au niveau conceptuel, chaque service est une fonction statique membre d'aucune classe, donc globale à l'application. La fonction de service peut recevoir un bloc de données en entrée et après traitement, fournir un bloc de données à envoyer. Cette stratégie propose donc que les services n'utilisent jamais directement les fonctionnalités du

socket. C’est plutôt la fonction qui appelle le service qui s’occupe de la réception, du passage des données d’entrées à la fonction de service, et par la suite du renvoie des données de sortie sur le *socket*. Cette technique permet d’obtenir une maintenance plus facile de l’application. Lorsqu’un service doit être ajouté, il suffit simplement de programmer la fonction exécutant le service au niveau de l’application serveur et de l’ajouter dans la liste des services disponibles. Elle peut ensuite être exécutée dès qu’un client en fait la demande. La stratégie d’exécution est résumée sur la figure suivante.

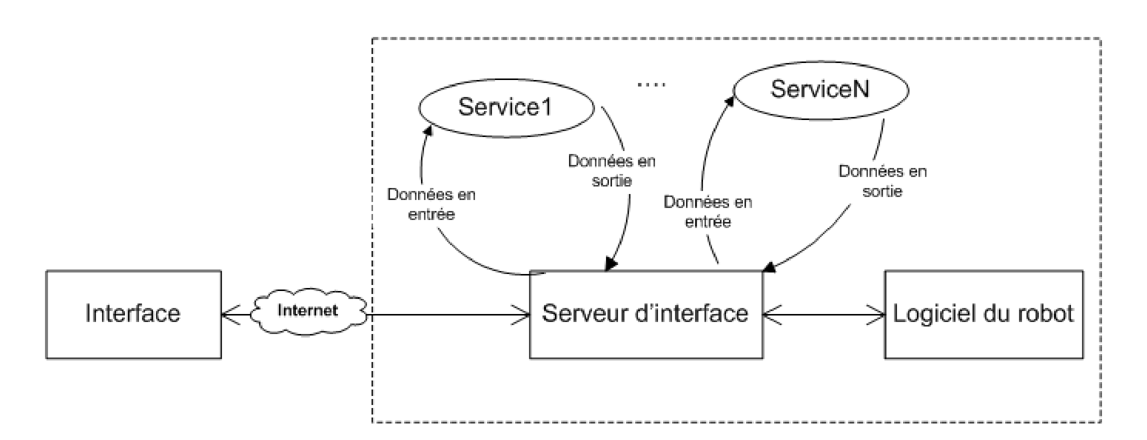


Figure 3.2 - Schéma de la stratégie d'exécution de services sur le serveur.

Avec une telle méthode de communication, les données sont transmises de façon brute, sans format particulier tel que XML puisque la fonction de l’interface qui exécute un service connaît le type de donnés à envoyer et le type de message à recevoir. Il n’y a donc aucun ajout de balises et les données peuvent être converties rapidement avec des fonctions de conversion de type.

Enfin, à un plus bas niveau de conception, cette méthode de communication suggère l’utilisation de *sockets* bloquants. Ceci signifie que lorsqu’un appel est fait sur la fonction de lecture des données du *socket*, cette fonction bloque l’exécution du programme tant et aussi longtemps que toutes les données ne sont pas reçues. On peut comprendre qu’une première interrogation nous vienne à l’esprit concernant l’impact de cette décision sur le comportement du logiciel d’interface client. Évidemment, il ne faut pas que, lorsqu’il y ait attente de données sur le *socket*, l’application gèle momentanément. Par chance, l’utilisation des *sockets* de la librairie graphique choisie permet d’utiliser des *sockets*

bloquant à l’exécution mais qui ne bloquent pas le reste de l’application graphique. Cela évite donc d’avoir à traiter les *sockets* dans des processus séparés.

4 Résultats

Dans ce chapitre, les résultats de l'étape de conception sont présentés à l'aide des diagrammes de classes des applications client et serveur. Ainsi, le lecteur pourra se familiariser avec la structure du programme client, c'est-à-dire l'interface usager et pourra comprendre la structure suggérée pour le modèle du serveur d'interface. Enfin, comme complément d'information, le contenu complet des classes est présenté en annexe.

4.1 Diagrammes de classes – Application client

Pour la structure des objets d'une application, le modèle UML propose la définition d'un diagramme de classes. Dans un souci de clarté, les diagrammes de classes présentés ne contiennent évidemment pas les attributs et méthodes de chacune des classes. [1]

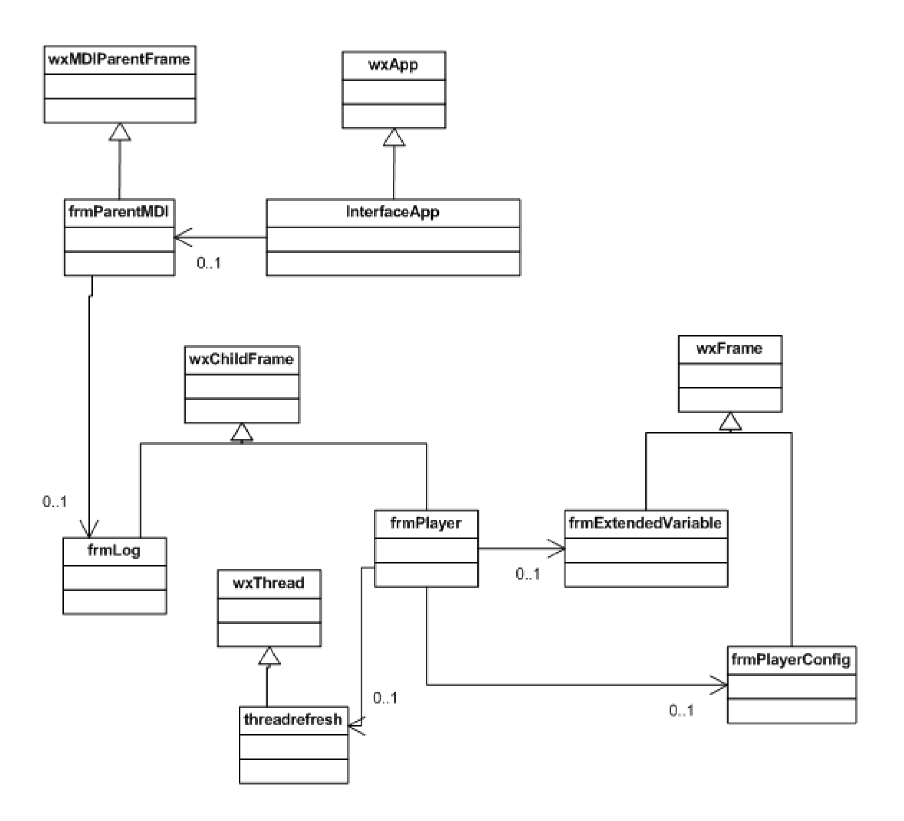


Figure 4.1 - Diagramme de classes - Interface client

[1] La description complète des classes, méthodes et attributs est fournie sur le CD-Rom d'accompagnement de ce document. Pour le contenu du CD-Rom, consultez l'annexe 1.

Le développement de l'interface usager s'est fait en profitant au maximum du contenu de la librairie wxWidgets. Chaque classe de l'application hérite d'une classe de la librairie graphique afin de personnaliser les fonctionnalités de base de la librairie. Cela fait en sorte que lorsqu'une entité telle que *frmPlayer* est instanciée cela résulte en une fenêtre manipulable par l'usager. Pour ajouter des objets de contrôle graphique tels que des boutons, des zones de texte ou des listes déroulantes, les classes créées doivent référer à d'autres objets de la librairie graphique. Notons que ces entités n'apparaissent pas sur le diagramme de classes, mais ils ont une importance capitale au fonctionnement de l'application. Par exemple, l'objet *wxSocket* utilisé dans la classe *frmPlayer* en est un de ceux-là. Sans l'utilisation de cet objet, la communication client/serveur aurait certainement été plus difficile et aurait probablement obligée l'utilisation de d'autres librairies spécialisées. La classe *frmPlayer* est certainement la classe centrale de l'application étant donné la quantité importante de fonctionnalités quelle contient. La majorité des évènements d'affichage et de fonctionnement sont directement relié à cette classe. À cause du modèle MDI sur lequel l'application a été développée, un usager peut ouvrir autant de fenêtres de joueurs qu'il désire. Ce qui ramène à dire, dans un sens plus fonctionnel, que l'application est capable de gérer une infinité de connexions vers les robots. La figure 4.2 présente l'objet *frmPlayer* une fois instancié par l'application ainsi que tous les objets créés par la classe, tels que des boutons ou des zones de texte.

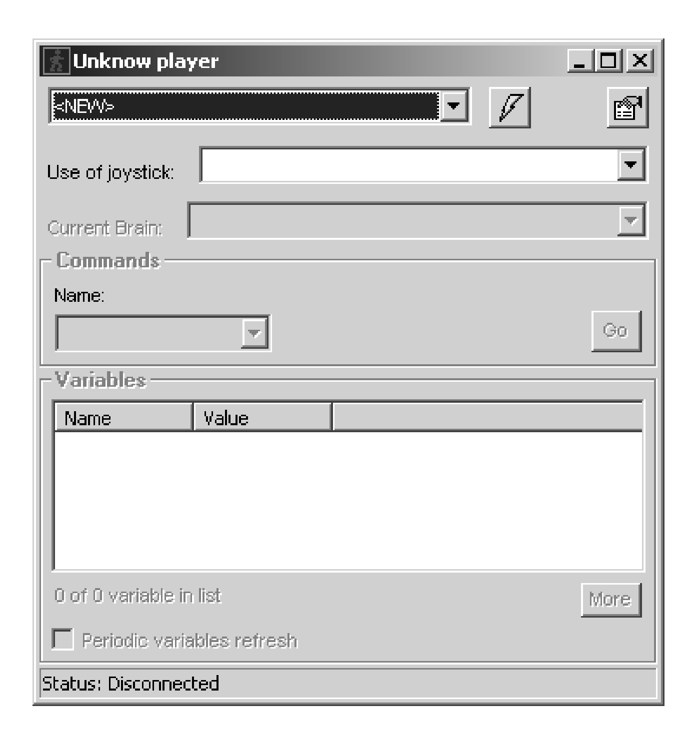


Figure 4.2 - L'objet frmPlayer est une fenêtre contant plusieurs contrôles wxWidgets

4.2 Utilisation de processus concurrents dans l’interface usager

Le diagramme de classes de la figure 4.1 contient aussi des classes qui ne sont pas visibles à l’affichage des fenêtres de l’application. C’est le cas notamment de la classe *threadrefresh* construite suite à l’héritage de la classe *wxThread.* L’utilisation de la classe *wxThread* facilite grandement la manipulation de processus concurrents qui agissent sur une même interface graphique. En utilisant cette technique à partir de la classe *threadRefresh*, cela nous permet d’obtenir un rafraîchissement des variables dans la fenêtre sans perturber le fonctionnement normal de l’application, comme cela aurait été le cas avec l’utilisation d’une minuterie *wxTimer*. Ainsi, dans la structure du programme, la classe *threadrefresh* n’a qu’une seule fonctionnalité : demander régulièrement à l’application serveur, via le service approprié, la valeur actuelle des variables afin de les afficher à l’interface. Néanmoins, il est important de bien comprendre que l’utilisation de processus concurrents dans une application entraîne inévitablement la venue de sections critiques et

de sémaphores d’exclusions mutuelles. C’est pourquoi la fonction de rafraîchissement des variables de bases est encadrée par un sémaphore de section critique puisqu’elle accède à des variables partagées entre le processus principal de la fenêtre du joueur et le processus de rafraîchissement des variables (*threadRefresh*). De plus, avec deux processus concurrents qui ont potentiellement accès au canal de communication en même temps, celui-ci doit être aussi protégé par un sémaphore d’exclusion mutuelle.

La limite du développement événementiel de la librairie *wxWidgets* est atteinte lorsqu’on ajoute la possibilité d’envoyer des commandes par manette de jeux. Encore une fois, la création d’un processus concurrent a été nécessaire pour résoudre la problématique causée par la programmation évènementielle, source d’instabilité dans la librairie. Dans notre cas, étant donné que tous les axes et les boutons de la manette de jeux sont potentiellement utilisables et que la fonction évènementielle proposée pour interpréter les boutons ne supporte pas plus de quatre boutons et un seul axe, l’utilisation d’une boucle de vérification de l’état de la manette de jeux devient inévitable. Sans utiliser de processus concurrent pour la réalisation de cette tâche, les mêmes instabilités apparaissent, ce qui influence le comportement de l’application. Ceci explique la raison d’être de la classe *threadJoystick*, qui a été utilisée pour résoudre cette problématique.

De plus, mentionnons que la limite théorique de la période des processus concurrents définie dans les classes *threadJoystick* et *threadRefresh* est de 100ms. Cette valeur maximale est fixée par les conditions du réseau de communication dont le délai de transmission peut parfois atteindre 100ms. Ce délai signifie qu’un paquet d’information peut prendre jusqu’à 100ms avant d’être acheminé de l’application serveur à l’interface client. Puisque ces processus concurrents utilisent une connexion réseaux pour transmettre des messages tels que l’état de la manette de jeux ou la valeur d’un lot de variable, il est utopique de croire que ces processus s’exécuteront toujours à l’intérieur de périodes plus courtes que 100ms.

4.3 Diagramme de classes du modèle serveur

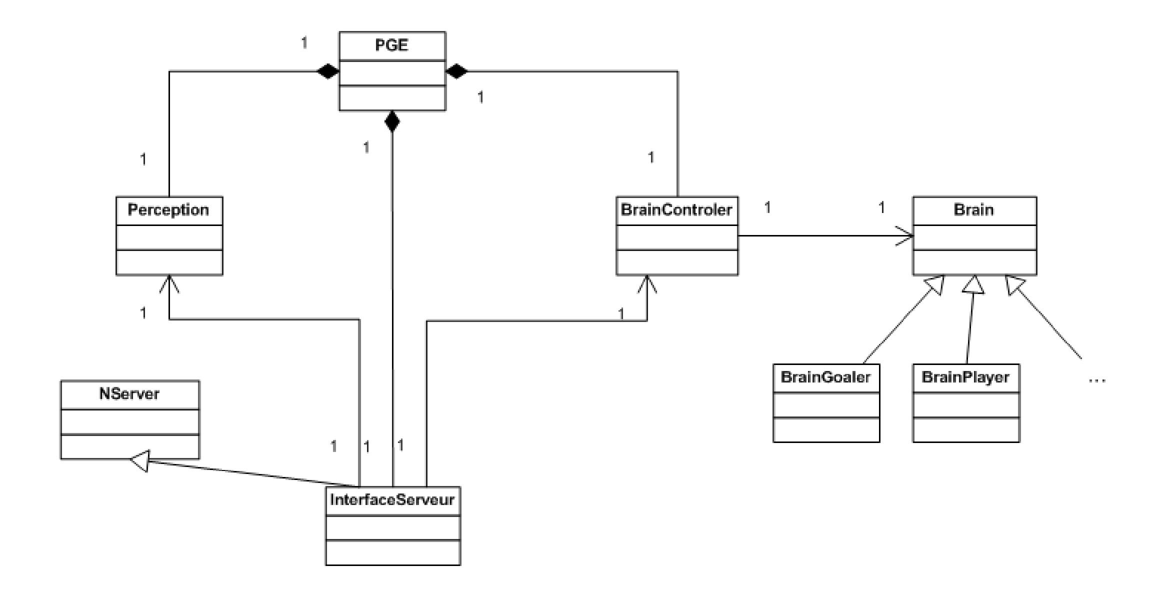


Figure 4.3 - Diagramme de classes – Serveur d'interface

Le diagramme de classes du serveur d'interface est un modèle de structure suggéré pour s'assurer de la meilleure compatibilité entre l'interface usager et le serveur des robots. Conceptuellement, la structure des classes du serveur d'interface est relativement simple. Le principal ajout à la structure qui existait déjà est la classe *InterfaceServer*, hérité à partir d'une classe *NServer* développée par Sylvain Marleau. Comme il est possible de le constater, le diagramme comporte peu de classes et les relations sont centrées sur la classe *InterfaceServer*. Celle-ci est l'élément clé du contact entre l'application client et le reste des objets du serveur. Dès l'instanciation des objets *BrainControler* et *Perception*, ceux-ci s'enregistrent auprès de la classe *InterfaceServer*. Ainsi, lorsqu'une communication est nécessaire vers un ou l'autre des objets à partir de l'interface client, comme lorsque que le client désire changer le cerveau courant, la classe *InterfaceServer* n'a qu'à interpréter le message et le diriger vers *BrainControler* qui s'occupera d'exécuter la tâche à l'aide de sa méthode public *SwitchBrain(int BrainID).*

En ce qui a trait aux classes cerveaux, il est possible de remarquer que tous les nouveaux cerveaux ajoutés dans l'application doivent hériter d'un cerveau de base. Cette

méthode de fonctionnement permet à l'objet *BrainControler* de ne jamais se soucier du type précis du cerveau en cours. L'ajout d'un cerveau spécialisé n'implique donc pas de modifications majeures à la classe contrôleur. Bref, l'interaction du contrôleur avec le cerveau courant se fait toujours par des méthodes connues de la classe de base. Il est important de comprendre que le contrôleur est le seul élément qui peut créer ou détruire un cerveau. Dès son instanciation, le *BrainControler* crée lui-même un cerveau de base qui répond à des commandes de façon inintelligente. Pour ajouter une touche de réflexion au robot, l'usager doit sélectionner un cerveau spécialisé à partir de l'interface client.

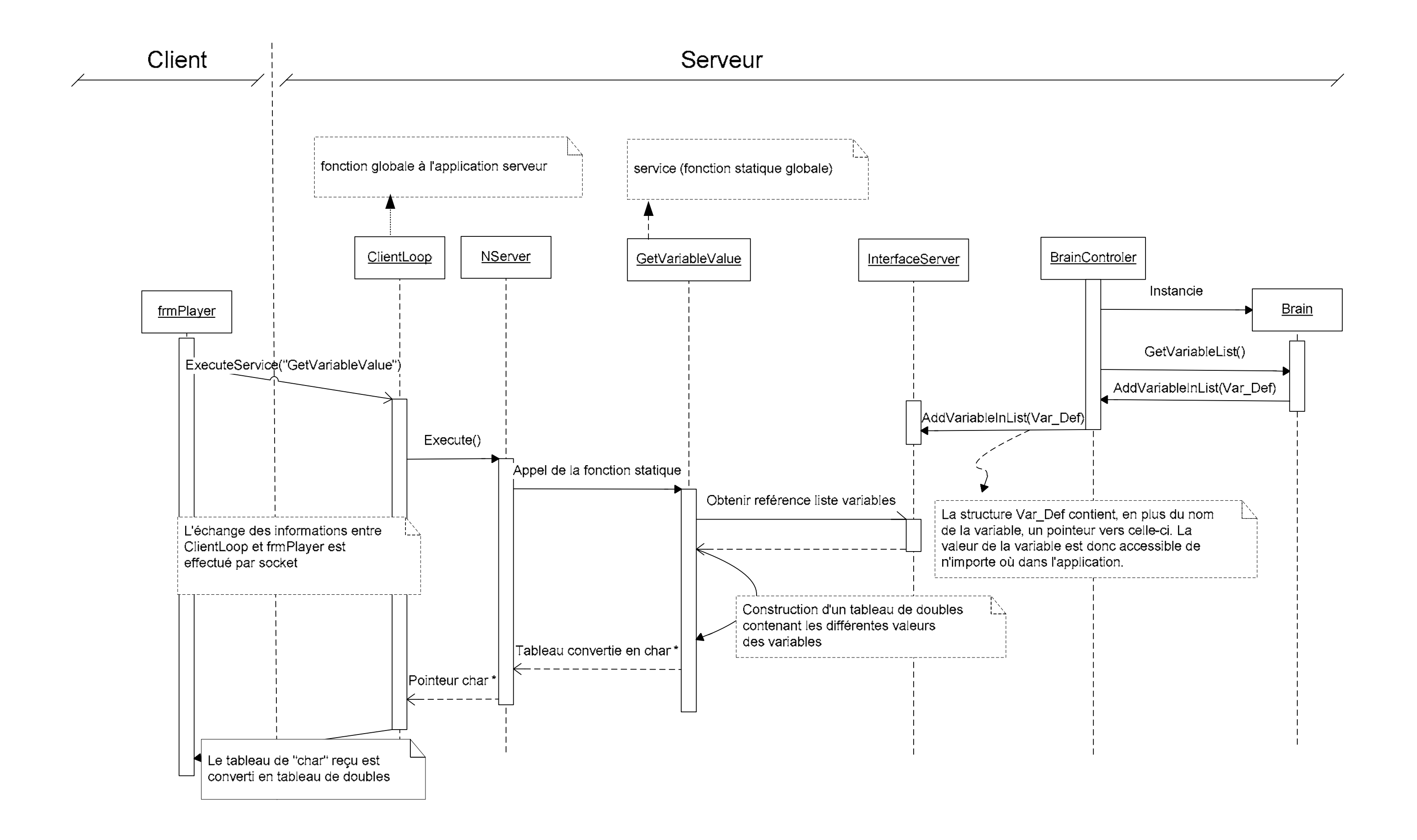


Figure 4.4 - Diagramme de séquence pour l'échange des valeurs de variables

La liste des variables disponibles à l'interface usager est définie par le cerveau courant. Lors de la création d'un objet cerveau, celui-ci envoie au *BrainControler* la liste des définitions des variables, contenant notamment le nom et un pointeur vers la variable. Ainsi, grâce au pointeur, les valeurs peuvent être accédées par n'importe quelle autre classe, sans nécessairement en faire la demande via une méthode du cerveau, et ce même si la variable est un attribut privé de la classe. Dans le diagramme de séquence présenté en figure 4.4, l'échange des valeurs de variables entres les différentes entités à partir du service *GetVariableValue* est schématisé. Cependant, il faut remarquer qu'avant d'être capable d'obtenir la valeur des variables, l'interface client doit connaître la liste des variables disponibles. Cette liste est obtenue suite à l'exécution du service *GetVariableDefinition*. Pour éviter une lourdeur inutile sur la figure précédente, l'exécution préalable de ce service n'est pas indiquée.

4.4 Utilisation de processus concurrent dans le serveur d'interface

Les processus concurrents sont davantage présents dans l'application serveur que dans l'application client. Sur le robot, plusieurs tâches doivent être traitées dans un semblant de parallèle et l'ajout du traitement d'une connexion constante avec l'interface client est une de ces tâches.

Le démarrage de l'application serveur entraîne la création de trois processus concurrents. Le premier processus, nommé *ClientLoop*, est celui qui écoute la réception des données sur le canal de communication. Dès que l'interface client envoie des données, la fonction *Read* du *socket* débloque et enregistre le message provenant de l'interface. Ensuite, le processus *ClientLoop* fait l'interprétation du message et lance le service approprié. Le deuxième processus concurrent, *BrainThread*, est traité par l'objet *Brain.* Ce processus est responsable d'appeler périodiquement la fonction *Think* du cerveau. Notons que *Think* est la fonction responsable des décisions prises par le robot. Afin d'obtenir un comportement précis, cette boucle est exécutée à toutes les 20ms. Enfin, le dernier processus est celui qui s'occupe des traitements de vision sur le robot. Rappelons que du

côté serveur, la librairie utilisée pour les processus concurrent est *MICROB*[2]. La figure suivante expose les différents processus concurrents en jeux, représenté par des ellipses, les autres objets constituants l'application serveur sont représentés par des rectangles.

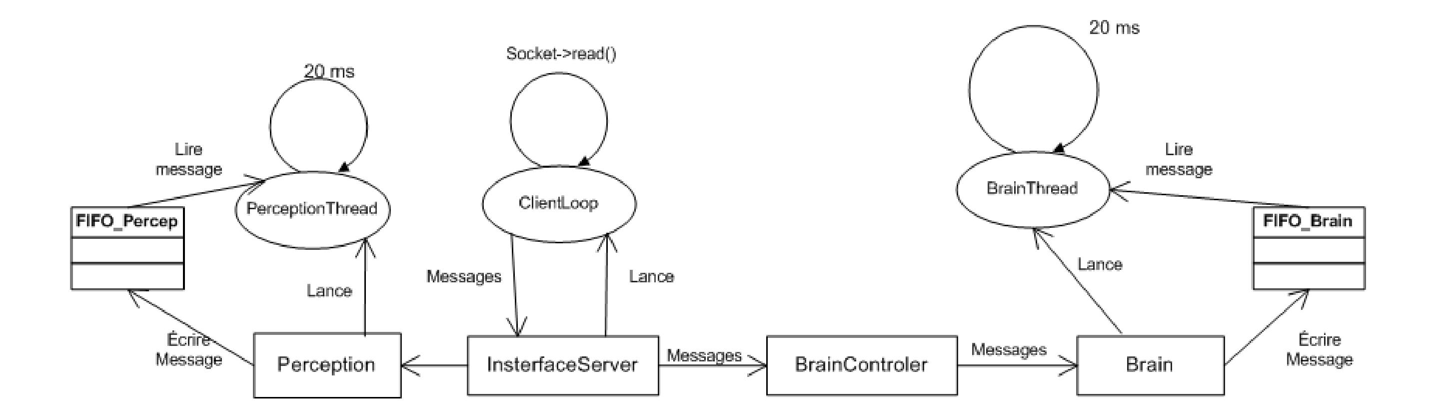


Figure 4.5 - Échange d'une commande à exécuter, sous forme de messages, entre deux processus gérés par des classes différentes.

La figure 4.5 indique sommairement de quelle façon une commande particulière est envoyée au cerveau. Par exemple, supposons qu'à partir de l'interface client, l'usager envoie la commande *GotoXY* avec deux paramètres donnant la position désirée du robot en X et Y sur la surface de jeux. Lorsque le message est reçu par le serveur, la fonction *ClientLoop* appelle le service *ExecuteSpecificCommand* qui s'arrange pour faire passer le message à travers la structure des classes connues. En bout de ligne, la commande voulue est placée dans une liste de type FIFO. Lorsque le processus *BrainThread* démarre sont exécution, il vérifie d'abord s'il n'y a pas de commandes à exécuter dans la liste avant d'appeler la fonction *Think*. Cette façon de faire permet d'interagir directement avec le robot sans nécessairement trop perturber son exécution.

4.5 Résultats de l'implantation de la stratégie de communication

Comme il a été mentionné dans le chapitre précédent, la méthodologie privilégiée pour l'échange d'informations entre les applications client et serveur est définie sous forme de

services. Les services développés pour combler les besoins d'information de l'interface usager sont les suivants :

- *GetNombreVariable*

  Ce service permet d'envoyer au client le nombre de variables disponibles dont il faut afficher la valeur à l'interface client.

- *GetVariableDefinition*

  Ce service envoie au client un tableau de structure *Variable_Def* de la taille équivalente au nombre de variables disponibles. Le tableau envoyé contient donc la définition des variables. Cette définition contient notamment le nom et le type (*int* ou *double*) de la variable.

- *GetVariableValue*

  Ce service permet d'envoyer la valeur des variables de base ou des variables étendues sous forme d'un tableau de double dont la taille dépend du nombre de variables dont il faut obtenir les valeurs.

- *GetNombreCommand*

  Ce service permet d'envoyer au client le nombre de commandes disponibles qu'il est possible de lancer à partir de l'interface client.

- *GetCommandDefinition*

  Ce service envoie au client un tableau de structure *Command_Def* de la taille équivalente au nombre de commandes disponibles. Le tableau envoyé contient donc la définition des commandes. Cette définition contient notamment un numéro d'identification de la commande, le nom de la commande, le nombre d'arguments attendus de cette commande ainsi que le nom de chacun des arguments.

- *ExecuteSpecificCommand*

Ce service permet de recevoir le numéro d'identification de la commande à exécuter sur le serveur ainsi que la valeur des arguments. La commande reçue est placée dans une liste de type FIFO nécessaire à la communication entre le *thread* du cerveau courant et le *thread* du serveur.

- *GetNombreBrain*

  Ce service permet d'envoyer au client le nombre de cerveaux disponibles qu'il est possible de changer dynamiquement à partir de l'interface client.

- *GetBrainDefinition*

  Ce service envoie au client un tableau de structure *Brain_Def* de la taille équivalente au nombre de cerveaux disponibles. Le tableau envoyé contient donc la définition des commandes. Cette définition contient notamment un numéro d'identification du cerveau et le nom du cerveau.

- *ChangeRunningBrain*

  Ce service reçoit un numéro, soit le numéro d'identification du cerveau désiré. Elle appelle ensuite les fonctions nécessaires pour détruire l'objet *Brain* présentement utilisé et pour construire le nouvel objet *Brain* désiré.

- *ExecuteJoystickCommand*

  Ce service permet d'interpréter les commandes reçues par la manette de jeux manipulée à partir l'interface client.

Lorsqu'un client se connecte au serveur via la fenêtre du joueur, le serveur hôte envoie la liste des services qu'il est capable de traiter. L'interface conserve la liste de ces services en mémoire pour la durée de la connexion. Puisque chaque fenêtre de joueur se connecte de façon indépendante, les serveurs ne sont pas obligés d'avoir la même version et la même liste de services. Seule la liste des services de base mentionnés précédemment est importante puisque toutes les fonctions de l'interface nécessitant une requête au serveur utilisent explicitement ces services.

4.6 Captures d'écran et manuel d'utilisateur

La section suivante contient un regroupement de capture d'écran de l'interface usager avec des explications supplémentaires permettant à l'usager de connaître le fonctionnement de l'application.

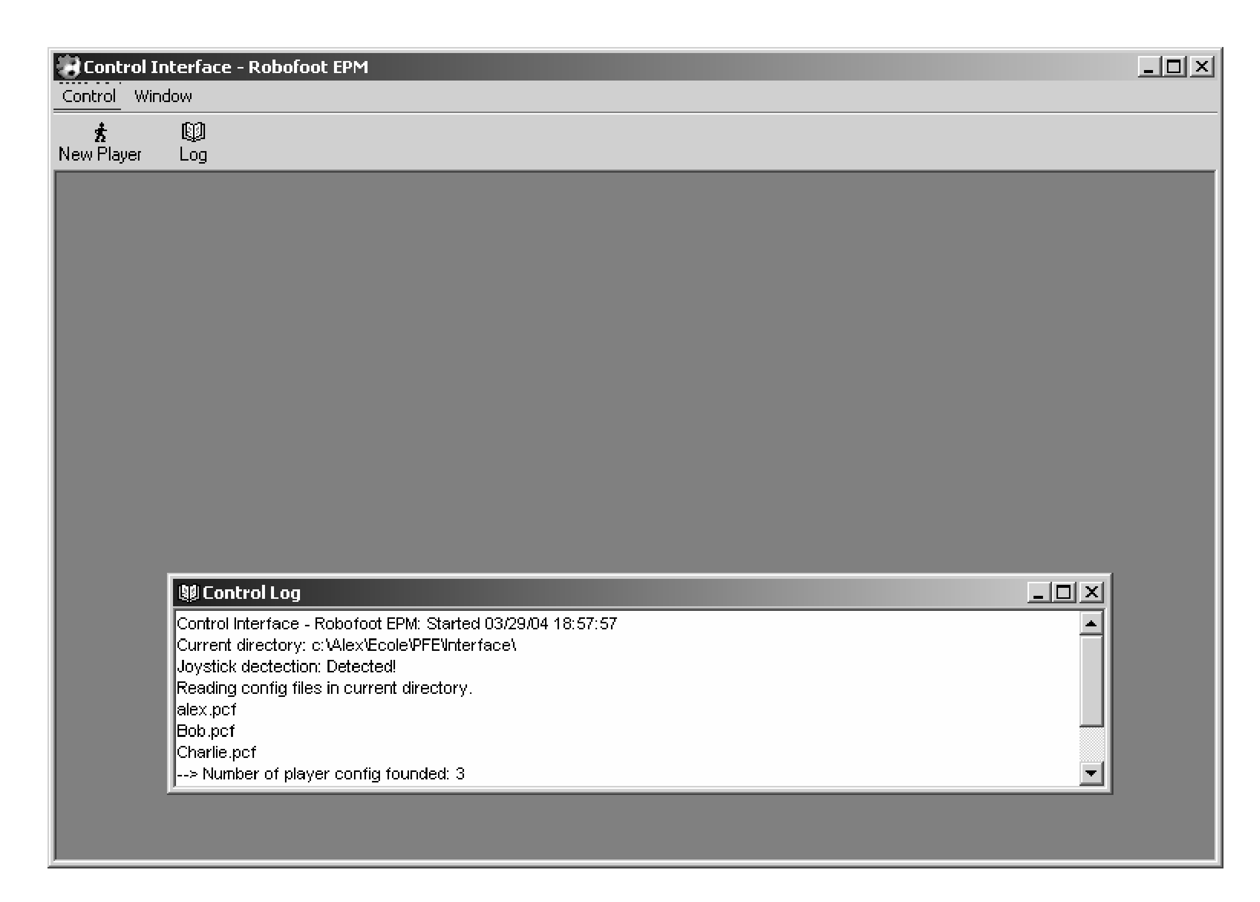


Figure 4.6 - Capture d'écran de la fenêtre de journal.

Suite au démarrage de l'application, l'utilisateur peut cliquer sur le bouton « Log » afin de faire apparaître une fenêtre contenant tous les messages d'information de l'application.

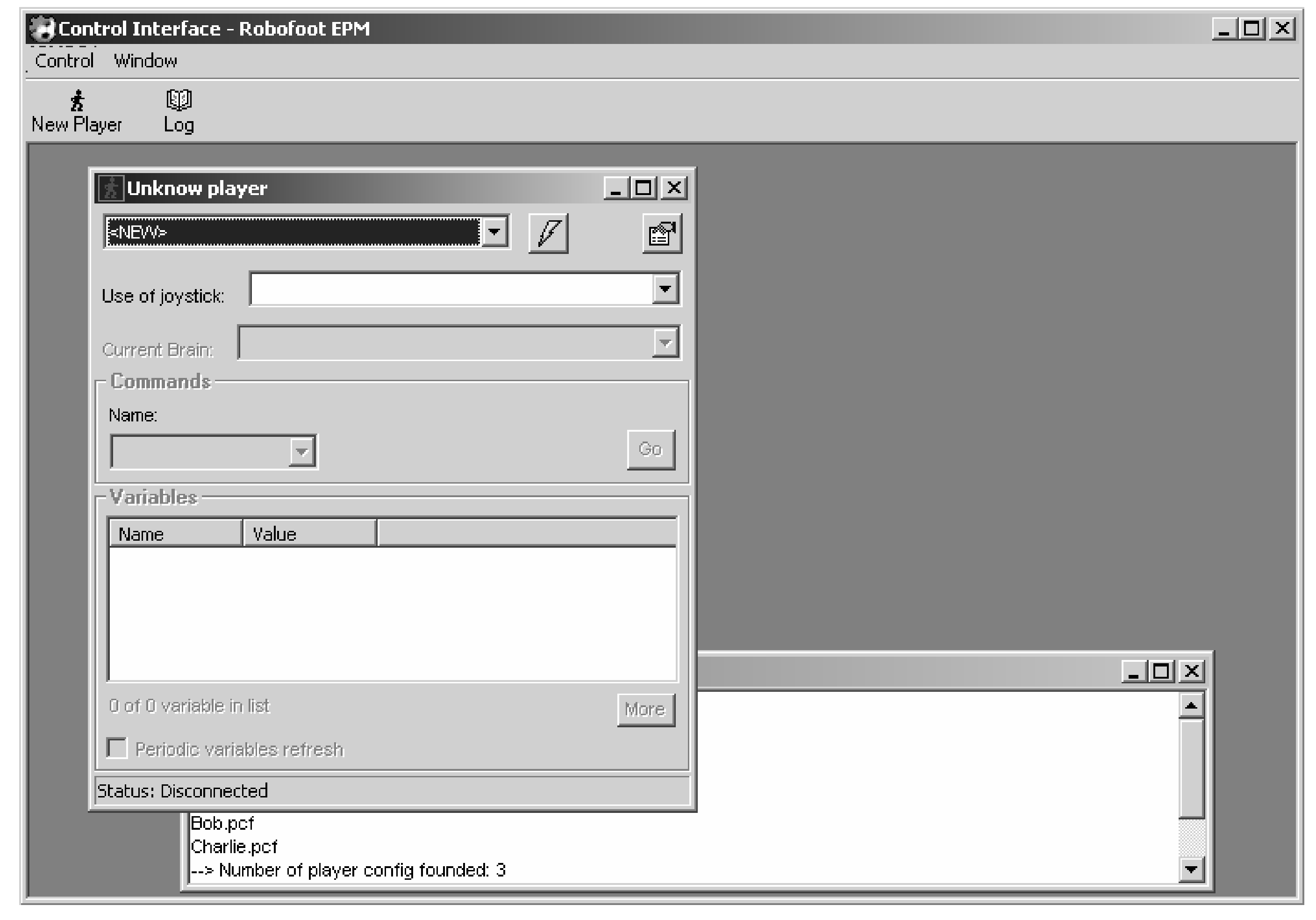


Figure 4.7 - Capture d'écran de la fenêtre déconnectée d'un joueur.

En cliquant sur le bouton « New Player », une fenêtre de contrôle du joueur apparaît. Évidemment, étant donné qu'aucune connexion n'est encore établie, plusieurs éléments de la fenêtre ne sont pas actifs. Remarquez la ressemblance graphique de l'application avec le prototype développé lors de l'étape de l'analyse (figure 3.1)

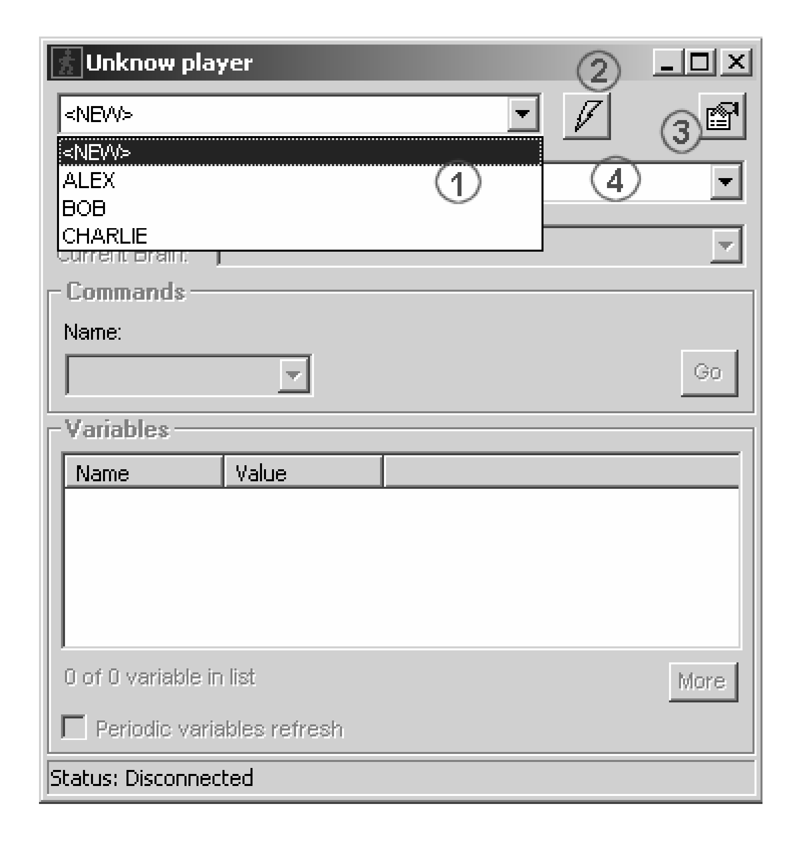


Figure 4.8 - Capture d’écran pour la description des fonctions lorsqu'aucune connexion n'est encore établie

1- **Liste déroulante contenant le nom des différentes connexions possibles**

2- **Bouton à cliquer pour établir une connexion avec le serveur sélectionné dans la liste**

3- **Bouton pour créer une nouvelle configuration lorsque la sélection est <NEW> ou pour modifier les attributs d’une configuration existante.**

4- **Liste déroulante pour activer l’utilisation d’une manette de jeux. Dans le cas où la sélection est « NO », la fenêtre ne pourra pas recevoir des commandes à envoyer au robot à l’aide d’une manette de jeux.**

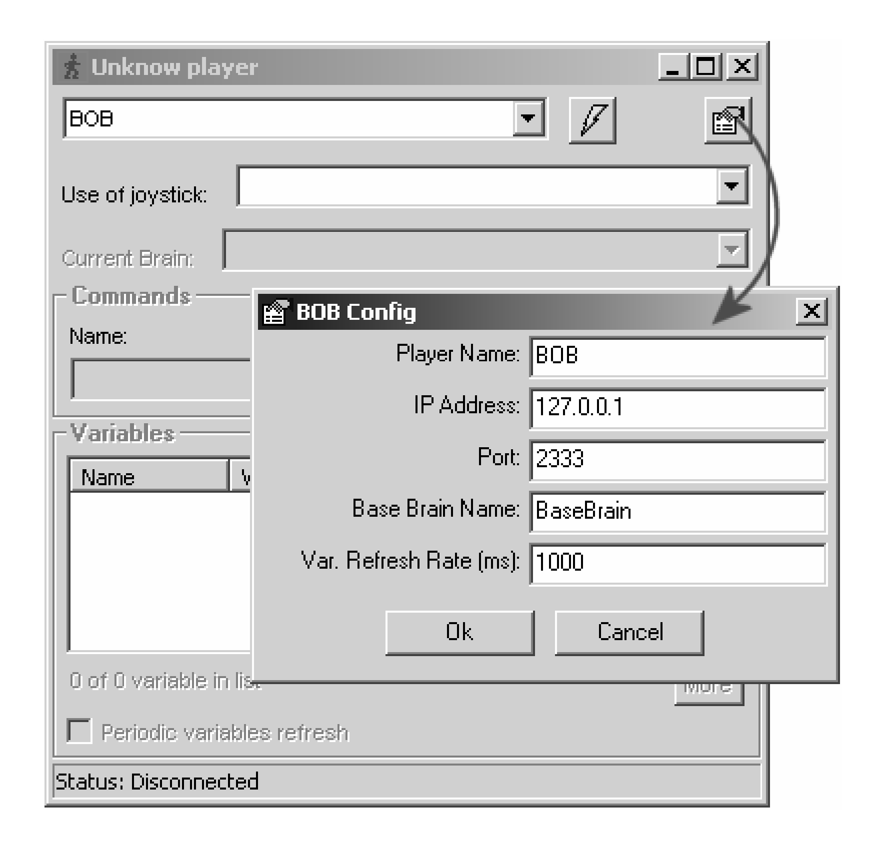


Figure 4.9 - Fenêtre de création d'une nouvelle configuration de connexion ou d'édition des paramètres d'une configuration existante.

Lorsque le paramètre *Var. Refresh Rate*, exprimé en milisecondes est fixé à 0, il n'y aura aucun rafraichissement périodique des variables de base pour toute la durée de la connexion.

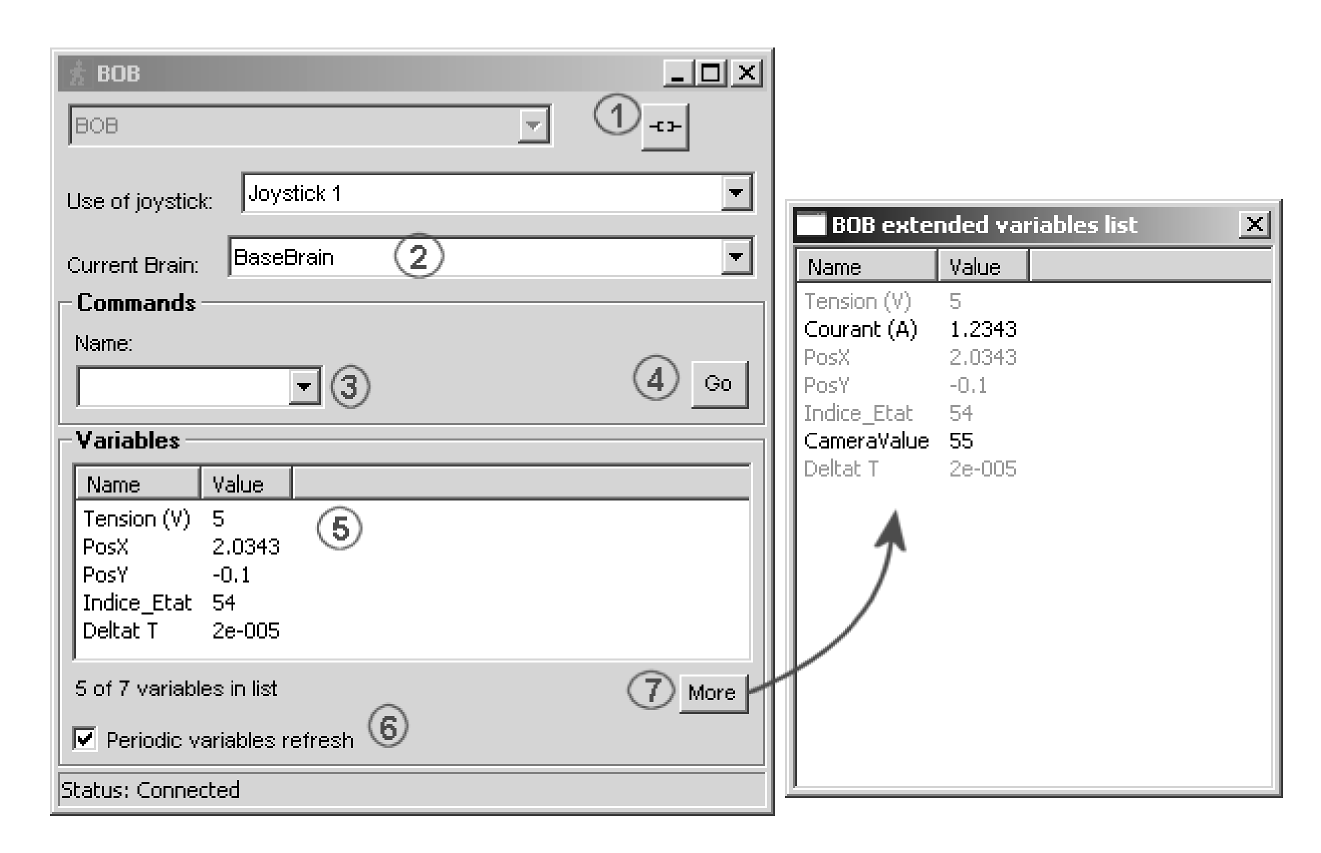


Figure 4.10 - Fenêtre de contrôle du joueur lorsqu'une connexion est correctement établie avec le serveur.

1- **Bouton pour fermer la connexion**

2- **Liste des cerveaux disponibles sur le serveur. La sélection d'une autre cerveau dans la liste a pour conséquence d'instancier un nouveau cerveau au niveau du robot (serveur) et de réinitialiser les listes de variables et de commandes au niveau de l'interface.**

3- **Liste des commandes possibles à envoyer au serveur.**

4- **Bouton d'action pour envoyer la commande et ses paramètres (s'il y a lieu) au serveur.**

5- **Liste des variables de base. Les variables et leur valeur apparaissent dans le même ordre que celui fournit par le cerveau actuellement instancié sur le serveur. La liste des valeurs de ces variables est rafraîchie à un taux défini comme paramètre à la configuration (voir figure 4.9).**

6- **Case à cocher pour arrêter ou redémarrer le processus concurrent de rafraîchissement des variables. Cette case à cocher n'est pas accessible lorsque le taux de rafraîchissement est fixé à 0.**

7- **Bouton permettant d'afficher la fenêtre des valeurs des variables étendues. Comparativement à la liste des variables de base, cette liste n'est pas rafraîchie périodiquement. Une requête de mise à jour est n'exécutée qu'au moment où l'usager clique sur le bouton.**

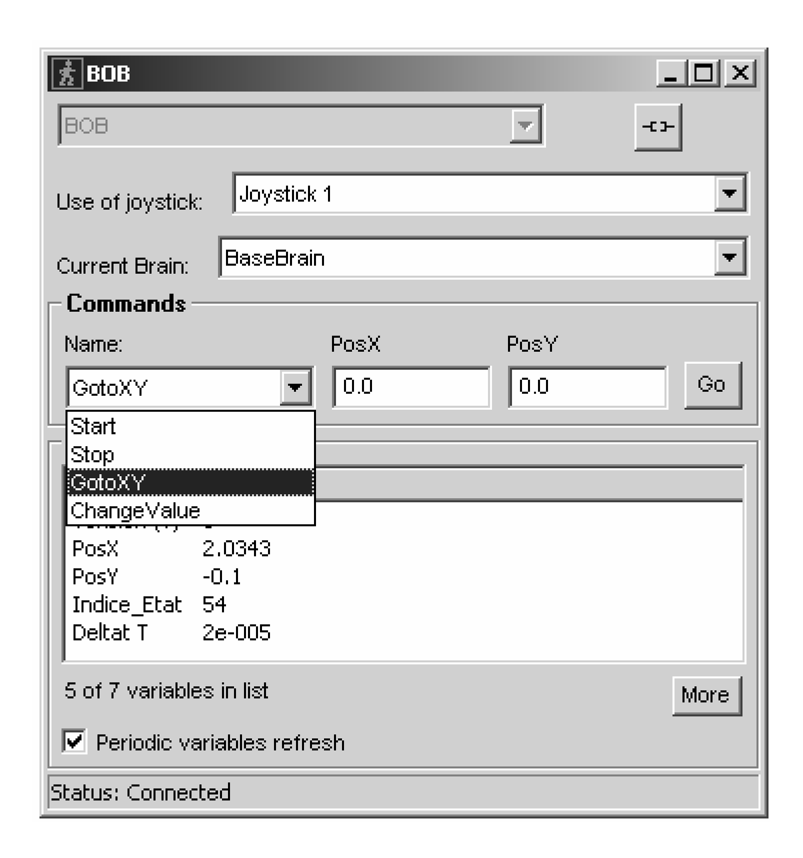


Figure 4.11 - Sélection d'une commande particulière à envoyer pour exécution.

Les paramètres varient (nombre et étiquette) selon la commande sélectionnée. La liste des commandes et des paramètres possibles est définie au niveau du serveur lors de la création du cerveau courant. Cette liste est envoyée à l'interface lors de l'étape d'initialisation de la connexion ou lors d'un changement de cerveau.

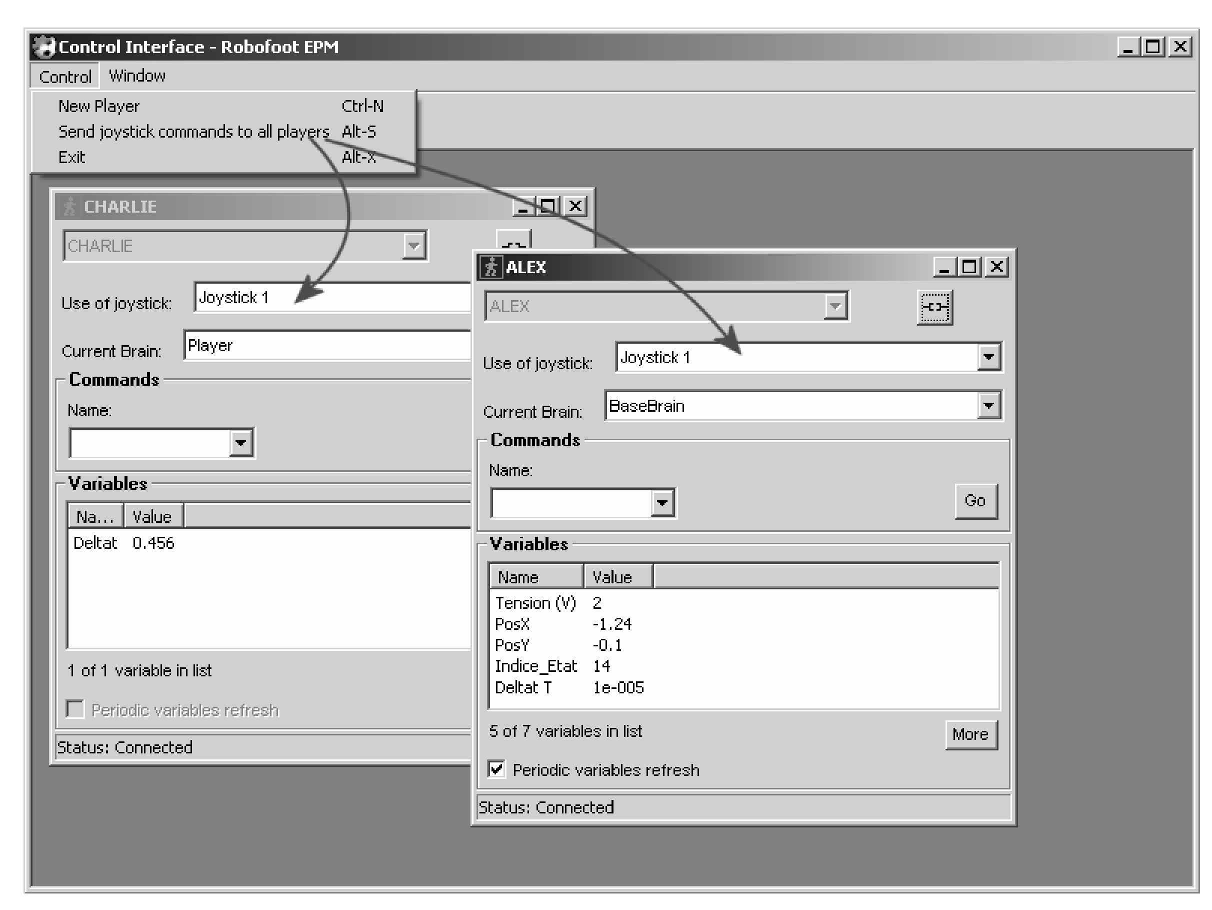


Figure 4.12 – Contenu du menu « Control »

Le menu cochable « Send joystick commands to all players » permet de soumettre les commandes perçues à partir de la manette de jeu vers toutes les fenêtres de connexion. Pour que la commande soit envoyée au robot, une fenêtre de connexion doit accepter l'utilisation de la manette de jeux à partir de la liste déroulante « Use of joystick ».

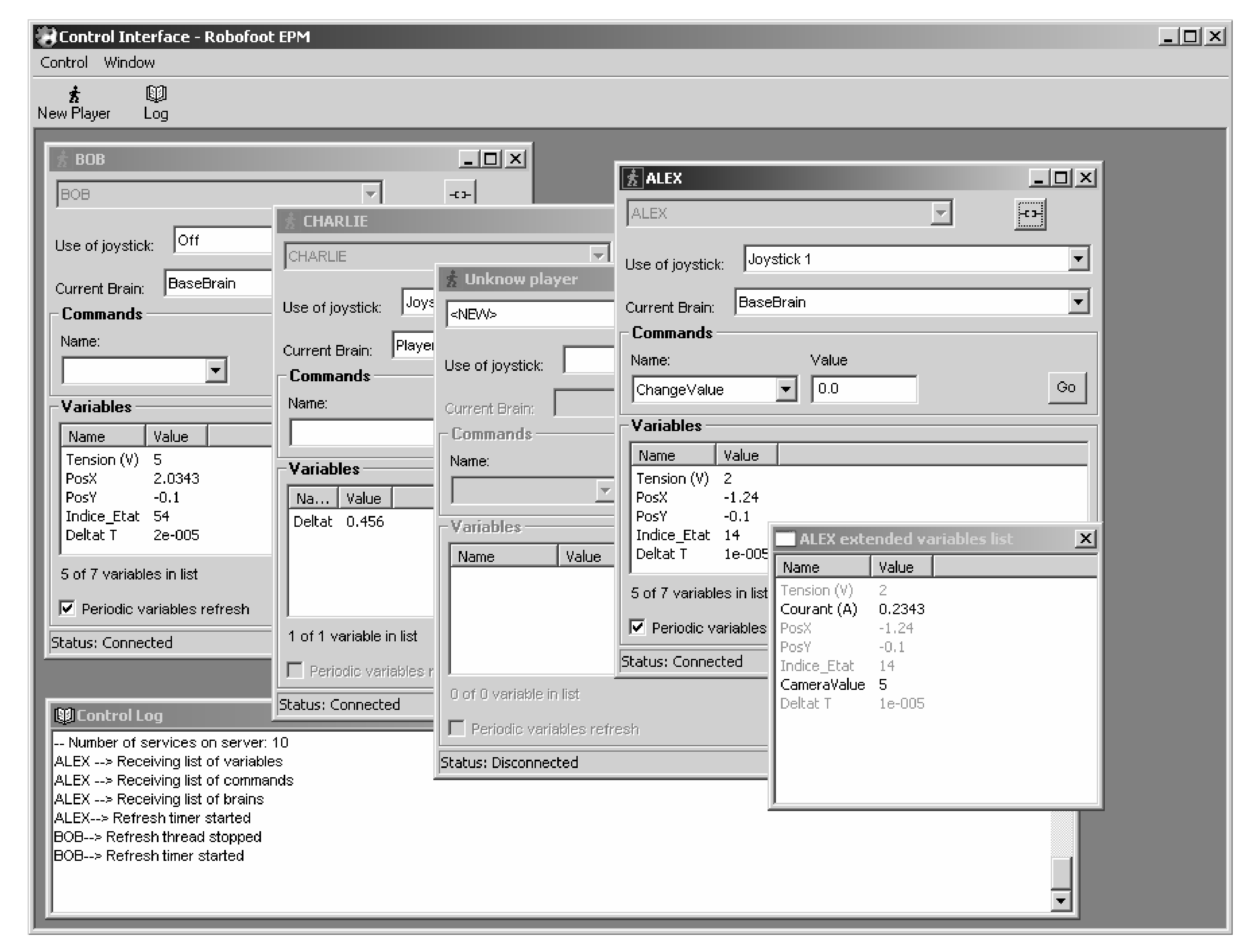


Figure 4.13 - Interface de contrôle en action manipulant sans problème plus de trois connexions simultanément.

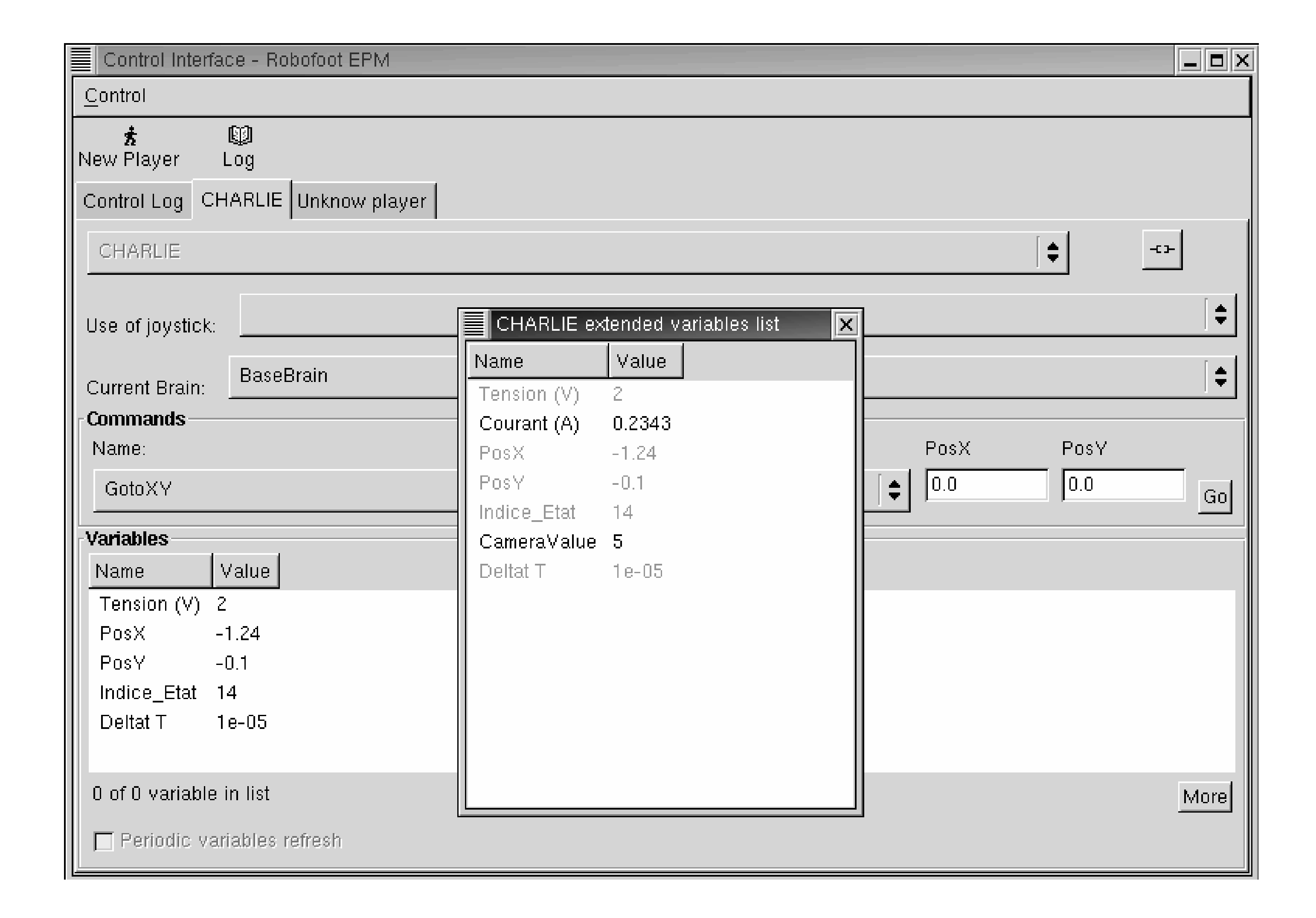


Figure 4.14 - Résultat de l'exécution de la même application après la compilation sous RedHat Linux avec GNOME et GTK+.

Nous remarquons facilement que l'aspect visuel de l'application est différent. Entre autre, les fenêtres MDI de Win32 sont remplacées par des onglets.

5 Discussion

Maintenant que le travail de développement est complété, il est pertinent d'effectuer une analyse critique de la méthodologie suivie ainsi que des résultats obtenus. De plus, quelques recommandations supplémentaires sont formulées concernant des possibilités de développements ultérieurs au projet. Enfin, une appréciation personnelle du projet conclue ce chapitre.

5.1 Commentaires sur la méthodologie utilisée

Le processus de développement logiciel suivi pour concevoir l'interface usager a donné de bons résultats. En prenant le temps de bien analyser les besoins du groupe au niveau de l'interface et des fonctionnalités, cela nous a permis de ne pas développer trop hâtivement une structure du logiciel qui serait devenue rapidement limitée pour des développements futurs. La facilité de modification d'une application de ce genre est très importante puisque les travaux sur le logiciel des robots sont en constantes évolutions. Par conséquent, la structure de classes développée pour l'interface client à fenêtrage MDI permet facilement d'intégrer d'autres fenêtres à l'application sans avoir à modifier les classes principales telles que *frmParentMDI* et *frmPlayer*. Pour le modèle de l'application serveur, le diagramme de classes proposé nous montre bien que l'implantation d'une telle architecture de classes ne nécessitera pas de changements majeurs dans la structure actuelle du logiciel des robots. Les éléments ajoutés concernent principalement les classes *InterfaceServer* et *BrainControler*.

Un deuxième élément de la méthodologie ayant donné de bons résultats concerne la conception préliminaire d'un prototype d'interface. Ce prototype à été bénéfique lorsqu'est venu le moment de dessiner l'interface avec les objets fenêtres de la librairie graphique choisie. Étant donnée la maîtrise limitée de la librairie wxWidgets en début de projet, un

gain considérable, particulièrement au niveau de la disposition des objets, a été réalisé grâce à ce développement. Malheureusement, il n'existe pas d'outils pour développer des interfaces graphiques rapidement qui donnent des résultats convenables avec la librairie wxWidgets. Par conséquent, la conception d'un prototype nous a simplement permis d'accélérer le choix du design et du comportement désiré de l'interface, sans accélérer la formulation des lignes de code nécessaire à la programmation des objets fenêtres.

L'utilisation du concept de services pour le traitement des communications entre l'application client et serveur s'est avérée une excellente idée. L'intégration logicielle des services est rapide, sans oublier qu'ils sont facilement modifiables. Cependant, il est impératif que le code de la fonction définissant un service soit sans faille. Dans le cas contraire, la connexion entre l'interface client et le serveur du robot pourrait s'interrompre momentanément. Ce désagrément peut aussi survenir lorsque le temps de traitement d'une fonction de service est trop important. Cela se traduit directement par un temps de réponse dépassant la limite, ayant pour conséquence une fermeture prématurée de la connexion.

5.2 Critique du choix de la librairie graphique

L'utilisation de la librairie graphique wxWidgets est inévitablement un élément clé de la réussite de ce projet. Sa facilité d'utilisation et les performances obtenues sont très semblables à d'autres librairies profitant d'un meilleur support telle que QT. Malgré cela, mentionnons que la librairie wxWidgets donne de bon résultat lorsque l'on connaît ses limites. Sauf que les limites rencontrées avec l'utilisation de la librairie wxWidgets sont mineures comparativement à l'immensité de ses possibilités, tant au niveau de la panoplie d'éléments de contrôles graphiques qu'au niveau des fonctionnalités non reliées aux développements d'interfaces graphiques telles que les *sockets* ou les processus concurrents. Normalement, la principale limite dans l'utilisation d'objets graphiques est fixée par le système d'exploitation ciblé. Par exemple, dans le projet actuel, l'utilisation d'une fenêtre MDI et de *sockets* limite de façon importante la portabilité de l'application à Win32 et Linux. [9]. Notons que la portabilité du projet courant vers Linux est en plus conditionnelle

à l’installation de GTK+. Par conséquent, pour s’assurer d’une plus grande portabilité de l’interface, il faudrait redessiner l’application en favorisant le développement par fenêtres indépendantes ou attendre que les classes concernées de la librairie subissent un développement plus poussé pour enrayer cette problématique. Cependant, en regardant les besoins spécifiques à notre application, il n’est pas nécessaire de cibler d’autres plateformes.

Donc, cette limite causée par l’utilisation de la librairie graphique n’est pas une contrainte au projet. Néanmoins, nous remarquons à la figure 4.14 que l’exécution de l’interface usager graphique sous Linux ne donne pas exactement les mêmes résultats que sous Win32. La fonctionnalité de l’application reste la même, mais l’esthétique est sensiblement différente. Les différences entre les interfaces graphiques ne sont malheureusement pas contrôlables et l’esthétisme est tributaire de la plateforme ciblée.

Souvent, les limites de la librairie sont causées par une incompréhension de son fonctionnement ou par la complexité de la fonction à réaliser. C’est notamment le cas avec la classe *wxTimer* qui est supposée fournir une exécution périodique d’une fonction. Dans le projet, ce type d’objet à été utilisé au départ afin d’implanter la fonctionnalité de rafraîchissement périodique des variables. Donc la solution initiale était d’utiliser un objet *wxTimer* croyant faussement qu’un processus concurrent en découlerait automatiquement à sa création. Malheureusement, l’utilisation d’un objet *wxTimer* introduit énormément d’instabilité dans l’application et a causé certains problèmes en bloquant régulièrement l’interface usager. Cette solution a été mise de côté au profit de l’utilisation de processus concurrents, qui sont garants d’une meilleure stabilité même s’il nécessite un peu plus de développement.

5.3 Commentaires sur les résultats obtenus

Suite à l’exécution de plusieurs tests de fonctionnement, les résultats obtenus sont plutôt satisfaisants. L’interaction entre l’interface de contrôle client et le modèle du serveur

est fluide et répond bien aux actions de l'utilisateur. Toutefois, à cause des multiples processus concurrents de l'application, tels que ceux responsables du rafraîchissement et celui permettant de vérifier l'état d'une manette de jeux, l'interface usager graphique peut devenir assez gourmande sur l'utilisation du processeur. Cela est particulièrement le cas lorsqu'il y a plusieurs connexions simultanées (quatre connexions ou plus)[2] vers des serveurs. Dans ce cas, l'interface usager répond moins bien et semble plutôt lente. Cela est évidemment causé par le fait que l'application doit négocier avec plusieurs éléments dynamiques en même temps comme la maintenance de plusieurs connexions et le suivi d'activités en parallèle sur ces connexions. Avec un processeur plus puissant, le comportement de l'interface, lorsqu'elle doit gérer plusieurs connexions, devrait être plus fluide.

5.4 Potentiel d'amélioration

L'interface de contrôle réalisée dans le cadre de ce projet est une application qui a un immense potentiel d'amélioration. Ces améliorations pourraient se concrétiser par l'ajout de nouvelles fonctionnalités ou par l'optimisation de fonctionnalités existantes. Une des fonctionnalités intéressantes qui aurait pu être développée dans ce projet est l'ajout d'une fenêtre OpenGL représentant en temps presque réel, la position des robots sur le terrain. La manipulation des robots pourrait ainsi se faire directement par des actions concrètes, directement à partir du modèle graphique incorporé à l'interface. D'autres ajouts, plus simples, pourrait aussi faciliter le contrôle des robots. En utilisant par exemple des touches de raccourcis clavier dans l'interface client, il serait possible de commander rapidement certaines fonctions des robots sans nécessiter la sélection d'une commande à l'aide de la souris.

À propos des possibilités d'optimisation, le contrôle actuel par manette de jeux propose l'utilisation d'un processus concurrent dont la période limite est de 100ms, fixée par les

[2] Les tests on été effectués sur PIII 650 256 Mo RAM

délais du réseau. Cette limite entraîne donc la perte de certaines informations lorsque l'état de la manette change dans un délai plus petit que 100ms. En ajoutant un processus de traitement plus rapide et en laissant à un autre processus la tâche d'envoyer les commandes sur le réseaux, cela permettrait probablement d'amélioration cette situation déplaisante. En résumé, ces ajouts ou modifications permettraient certainement d'obtenir une application de meilleure qualité et aussi plus performante.

6 Références bibliographiques

6.1 Ressources électroniques

[1] GTK + - The GIMP Toolkit
http://www.gtk.org/ , consulté le 4 avril 2004

[2] MICROB : Modules Integrés pour le Contrôle de ROBots
http://www.robotique.ireq.ca/microb/fr/index.html, consulté le 4 avril 2004

[3] Office québécois de la langue française, Grand dictionnaire terminologique
http://www.grandictionnaire.com, consulté le 4 avril 2004

[4] RedHat – Linux, Embedded Linux and Open Source Solution
http://www.redhat.com/, consulté le 4 avril 2004

[5] RoboCup official Site
http://www.robocup.org/ , consulté le 4 avril 2004

[6] Robofoot ÉPM
http://robofoot.auto.polymtl.ca/accueil.html, consulté le 4 avril 2004

[7] Trolltech – Creators of Qt – The multi-plateform C++ GUI/API
http://www.trolltech.com/, consulté le 4 avril 2004

[8] Visual Basic Developer Center, Microsoft Corporation
http://msdn.microsoft.com/vbasic/, consulté le 4 avril 2004

[9] wxWidgets Home
http://www.wxwindows.org/, consulté le 4 avril 2004

[10] wxWigets – Supported classes for each port
http://www.wxwindows.org/supported.htm, consulté le 4 avril 2004

6.2 Ressources Papier

[11] DANESH, Arman, *Mastering Linux Premium Edition,*
Sybex, 1999, p9 (Linux as Free Software).

[12] DEITEL, DEITEL, *Comment programmer en C++*, 2e édition,
Les éditions Reynald Goulet inc., 1998, 1116p.

[13] Larman, C. *Applying UML and Patterns - An Introduction to Object-Oriented Analysis and Design*. Prentice Hall. 1998. ISBN 0-13-748880-7

7 Annexe

7.1 Annexe 1 – Contenu du CD-Rom

- **Interface** – Répertoire des fichiers projets pour interface client
    - **src :** fichiers sources .h, .cpp et makefile
    - **doc :** documentation en format HTML (voir fichier index.html)

- **Interface_Server** – Répertoire des fichiers projets pour modèle serveur
    - **src :** fichiers sources .h, .cpp
    - **doc :** documentation en format HTML (voir fichier index.html)

- **Prototype_VB** – Répertoire des fichiers projets VB 6.0 pour le prototype

- **Rapport** – Répertoire contenant ce rapport.